# Automated processor generation system for designing a configurable processor and method for the same

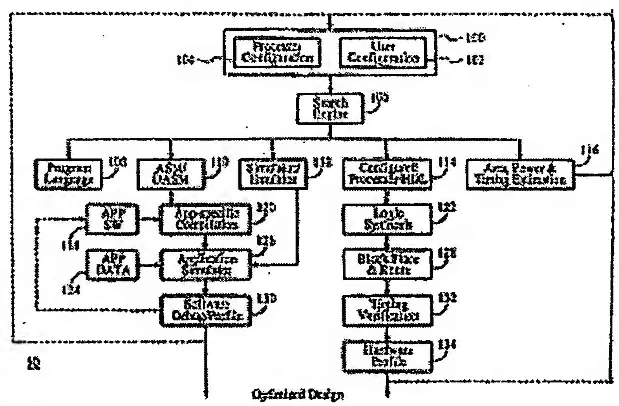
| | | |
|---|---|---|
| **Patent number:** | TW539965B | |
| **Publication date:** | 2003-07-01 | |
| **Inventor:** | KILLIAN EARL A (US); GONZULEZ RICARDO E (MX); DIXIT ASHISH B (US); LAM MONICA (US); LICHTENSTEIN WALTER D (US) | |
| **Applicant:** | TENSILICA INC (US) | |
| **Classification:** | | |
| **- international:** | ***G06F11/28; G06F9/45; G06F17/50;*** **G06F11/28; G06F9/45; G06F17/50;** (IPC1-7): G06F17/50 | |
| **- european:** | G06F17/50D | |
| **Application number:** | TW20000102150 20000310 | |
| **Priority number(s):** | US19990246047 19990205; US19990323161 19990527; US19990322735 19990528 | |

**Also published as:**

- WO0046704 (A3)
- WO0046704 (A3)
- EP1159693 (A3)
- EP1159693 (A2)
- EP1159693 (A0)


Abstract of **TW539965B**

In a first aspect of the invention, a configurable RISC processor implements an instruction set which provides good code density in a fixed-length high-performance encoding based on RISC principles, including a general register with load/store architecture. Further, the processor implements a simple variable-length encoding that maintains high performance. In a second aspect of the invention, when selecting and building a processor configuration, a user creates a new set of user-defined instructions, places them in a file directory, and invokes a tool that processes the user instructions and transforms them into a form usable by the software development tools. In this way, the user may customize a processor configuration by adding new instructions and within minutes, be able to evaluate that feature. The user is able to keep multiple sets of potential instructions and easily switch between them when evaluating their application. In a third aspect of the invention, an automated processor design tool uses a description of customized processor instruction set extensions in a standardized language to develop a configurable definition of a target instruction set, a hardware description language description of circuitry necessary to implement the instruction set, and development tools which can be used to develop applications for the processor and to verify it. The standardized language is capable of handling instruction set extensions which modify processor state or use configurable processors. By providing a constrained domain of extensions and optimizations, the process can be automated to a high degree, thereby facilitating fast and reliable development.



8P102150

# AUTOMATED PROCESSOR GENERATION SYSTEM FOR DESIGNING A CONFIGURABLE PROCESSOR AND METHOD FOR THE SAME

## BACKGROUND OF THE INVENTION

### 1. Field of the Invention

The present invention is directed to microprocessor systems; more particularly, the invention is directed to the design of an application solution containing one or more processors where the processors in the system are configured and enhanced at the time of their design to improve their suitability to a particular application. The invention is additionally directed to a system in which application developers can rapidly develop instruction extensions, such as new instructions, to an existing instruction set architecture, including new instruction which manipulate user-defined processor state, and immediately measure the impact of the extension to the application run time and to the processor cycle time.

### 2. Description of Related Art

Processors have traditionally been difficult to design and to modify. For this reason, most systems that contain processors use ones that were designed and verified once for general-purpose use, and then used by multiple applications over time. As such, their suitability for a particular application is not always ideal. It would often be appropriate to modify the processor to execute a particular application's code better (e.g., to run faster, consume less power, or cost less). However, the difficulty, and therefore the time, cost, and risk of even modifying an existing processor design is high, and this is not typically done.

To better understand the difficulty in making a prior art processor configurable, consider its development. First, the instruction set architecture (ISA) is developed. This is a step which is essentially done once and used for decades by many systems. For example, the Intel Pentium® processor can trace the legacy of its instruction set back to the 8008 and 8080 microprocessors introduced in the mid-1970's. In this process, based on predetermined ISA design criteria, the ISA instructions, syntax, etc. are developed, and software development tools for that ISA such as assemblers, debuggers, compilers and the like are developed. Then, a simulator for that particular ISA is developed and various benchmarks are run to evaluate the effectiveness of the ISA and the ISA is revised according to the results of the evaluation. At some point, the ISA will be considered satisfactory, and the ISA process will end with a fully developed ISA specification, an ISA simulator, an ISA verification suite and a development suite including, e.g., an assembler, debugger, compiler, etc.

Then, processor design commences. Since processors can have useful lives of a number of years, this process is also done fairly infrequently -- typically, a processor will be designed once and used for many years by several systems. Given the ISA, its verification suite and simulator and various processor development goals, the microarchitecture of the processor is designed, simulated and revised. Once the microarchitecture is finalized, it is implemented in a hardware description language (HDL) and a



microarchitecture verification suite is developed and used to verify the HDL implementation (more on this later). Then, in contrast to the manual processes described to this point, automated design tools may synthesize a circuit based on the HDL description and place and route its components. The layout may then be revised to optimize chip area usage and timing. Alternatively, additional manual processes may be used to create a floorplan based on the HDL description, convert the HDL to circuitry and then both manually and automatically verify and lay the circuits out. Finally, the layout is verified to be sure it matches the circuits using an automated tool and the circuits are verified according to layout parameters.

✓ After processor development is complete, the overall system is designed. Unlike design of the ISA and processor, system design (which may include the design of chips that now include the processor) is quite common and systems are typically continuously designed. Each system is used for a relatively short period of time (one or two years) by a particular application. Based on predetermined system goals such as cost, performance, power and functionality; specifications of pre-existing processors; specifications of chip foundries (usually closely tied with the processor vendors), the overall system architecture is designed, a processor is chosen to match the design goals, and the chip foundry is chosen (this is closely tied to the processor selection).

Then, given the chosen processor, ISA and foundry and the simulation, verification and development tools previously developed (as well as a standard cell library for the chosen foundry), an HDL implementation of the system is designed, a verification suite is developed for the system HDL implementation and the implementation is verified. Next, the system circuitry is synthesized, placed and routed on circuit boards, and the layout and timing are re-optimized. Finally, the boards are designed and laid out, the chips are fabricated and the boards are assembled.

Another difficulty with prior art processor design stems from the fact that it is not appropriate to ✓ simply design traditional processors with more features to cover all applications, because any given application only requires a particular set of features, and a processor with features not required by the application is overly costly, consumes more power and is more difficult to fabricate. In addition it is not possible to know all of the application targets when a processor is initially designed. If the processor modification process could be automated and made reliable, then the ability of a system designer to create application solutions would be significantly enhanced.

As an example, consider a device designed to transmit and receive data over a channel using a complex protocol. Because the protocol is complex, the processing cannot be reasonably accomplished entirely in hard-wired, e.g., combinatorial, logic, and instead a programmable processor is introduced into the system for protocol processing. Programmability also allows bug fixes and later upgrades to protocols to be done by loading the instruction memories with new software. However, the traditional processor was probably not designed for this particular application (the application may not have even existed when the processor was designed), and there may be operations that it needs to perform that require many instructions to accomplish which could be done with one or a few instructions with additional processor logic.

Because the processor cannot easily be enhanced, many system designers do not attempt to do so, and instead choose to execute an inefficient pure-software solution on an available general-purpose processor. The inefficiency results in a solution that may be slower, or require more power, or be costlier (e.g., it may require a larger, more powerful processor to execute the program at sufficient speed). Other designers choose to provide some of the processing requirements in special-purpose hardware that they design for the application, such as a coprocessor, and then have the programmer code up access to the special-purpose hardware at various points in the program. However, the time to transfer data between the processor and such special-purpose hardware limits the utility of this approach to system optimization because only fairly large units of work can be sped up enough so that the time saved by using the special-purpose hardware is greater than the additional time required to transfer data to and from the specialized hardware.

In the communication channel application example, the protocol might require encryption, error-correction, or compression/decompression processing. Such processing often operates on individual bits rather than a processor's larger words. The circuitry for a computation may be rather modest, but the need for the processor to extract each bit, sequentially process it and then repack the bits adds considerable overhead.

As a very specific example, consider a Huffman decode using the rules shown in TABLE I (a similar encoding is used in the MPEG compression standard). Both the value and the

| Pattern | Value | Length |
|---|---|---|
| 0 0 X X X X X X | 0 | 2 |
| 0 1 X X X X X X | 1 | 2 |
| 1 0 X X X X X X | 2 | 2 |
| 1 1 0 X X X X X | 3 | 3 |
| 1 1 1 0 X X X X | 4 | 4 |
| 1 1 1 1 0 X X X | 5 | 5 |
| 1 1 1 1 1 0 X X | 6 | 6 |
| 1 1 1 1 1 1 0 X | 7 | 7 |
| 1 1 1 1 1 1 1 0 | 8 | 8 |
| 1 1 1 1 1 1 1 1 | 9 | 8 |

TABLE I

length must be computed, so that `length` bits can be shifted off to find the start of the next element to be decoded in the stream.

There are a multitude of ways to code this for a conventional instruction set, but all of them require many instructions because there are many tests to be done, and in contrast with a single gate delay



for combinatorial logic, each software implementation requires multiple processor cycles. For example, an efficient prior art implementation using the MIPS instruction set might require six logical operations, six conditional branches, an arithmetic operation, and associated register loads. Using an advantageously-designed instruction set the coding is better, but still expensive in terms of time: one logical operation, six conditional branches, an arithmetic operation and associated register loads.

In terms of processor resources, this is so expensive that a 256-entry lookup table is typically used instead of coding the process as a sequence of bit-by-bit comparisons. However, a 256-entry lookup table takes up significant space and can be many cycles to access as well. For longer Huffman encodings, the table size would become prohibitive, leading to more complex and slower code.

A possible solution to the problem of accommodating specific application requirements in processors is to use configurable processors having instruction sets and architectures which can be easily modified and extended to enhance the functionality of the processor and customize that functionality. Configurability allows the designer to specify whether or how much additional functionality is required for her product. The simplest sort of configurability is a binary choice: either a feature is present or absent. For example, a processor might be offered either with or without floating-point hardware.

Flexibility may be improved by configuration choices with finer gradation. The processor might, for example, allow the system designer to specify the number of registers in the register file, memory width, the cache size, cache associativity, etc. However, these options still do not reach the level of customizability desired by system designers. For example, in the above Huffman decoding example, although not known in the prior art the system designer might like to include a specific instruction to perform the decode, e.g.,

```
huff8 t1, t0
```

where the most significant eight bits in the result are the decoded value and the least significant eight bits are the length. In contrast to the previously described software implementation, a direct hardware implementation of the Huffman decode is quite simple -- the logic to decode the instruction represents roughly thirty gates for just the combinatorial logic function exclusive of instruction decode, etc., or less than 0.1% of a typical processor's gate count, and can be computed by a special-purpose processor instruction in a single cycle, thus representing an improvement factor of 4-20 over using general-purpose instructions only.

Prior art efforts at configurable processor generation have generally fallen into two categories: logic synthesis used with parameterized hardware descriptions; and automatic retargeting of compilers and assemblers from abstract machine descriptions. In the first category fall synthesizable processor hardware designs such as the Synopsys DW8051 processor, the ARM/Synopsys ARM7-S, the Lexra LX-4080, the ARC configurable RISC core; and to some degree the Synopsys synthesizable/configurable PCI bus interface.

Of the above, the Synopsys DW8051 includes a binary-compatible implementation of an existing processor architecture; and a small number of synthesis parameters, e.g., 128 or 256 bytes of internal

RAM, a ROM address range determined by a parameter `rom_addr_size`, an optional interval timer, a variable number (0-2) of serial ports, and an interrupt unit which supports either six or thirteen sources. Although the DW8051 architecture can be varied somewhat, no changes in its instruction set architecture are possible.

The ARM/Synopsys ARM7-S processor includes a binary-compatible implementation of existing architecture and microarchitecture. It has two configurable parameters: the selection of a high-performance or low-performance multiplier, and inclusion of debug and in-circuit emulation logic. Although changes in the instruction set architecture of the ARM7-S are possible, they are subsets of existing non-configurable processor implementations, so no new software is required.

The Lexra LX-4080 processor has a configurable variant of the standard MIPS architecture and has no software support for instruction set extensions. Its options include a custom engine interface which allows extension of MIPS ALU opcodes with application-specific operations; an internal hardware interface which includes a register source and a register or 16 bit-wide immediate source, and destination and stall signals; a simple memory management unit option; three MIPS coprocessor interfaces; a flexible local memory interface to cache, scratchpad RAM or ROM; a bus controller to connect peripheral functions and memories to the processor's own local bus; and a write buffer of configurable depth.

The ARC configurable RISC core has a user interface with on-the-fly gate count estimation based on target technology and clock speed, instruction cache configuration, instruction set extensions, a timer option, a scratch-pad memory option, and memory controller options; an instruction set with selectable options such as local scratchpad RAM with block move to memory, special registers, up to sixteen extra condition code choices, a 32 x 32 bit scoreboarded multiply block, a single cycle 32 bit barrel-shifter/rotate block, a normalize (find first bit) instruction, writing results directly to a command buffer (not to the register file), a 16 bit MUL/MAC block and 36 bit accumulator, and sliding pointer access to local SRAM using linear arithmetic; and user instructions defined by manual editing of VHDL source code. The ARC design has no facility for implementing an instruction set description language, nor does it generate software tools specific to the configured processor.

The Synopsys configurable PCI interface includes a GUI or command line interface to installation, configuration and synthesis activities; checking that prerequisite user actions are taken at each step; installation of selected design files based on configuration (e.g., Verilog vs. VHDL); selective configuration such as parameter setting and prompting of users for configuration values with checking of combination validity, and HDL generation with user updating of HDL source code and no editing of HDL source files; and synthesis functions such as a user interface which analyzes a technology library to select I/O pads, technology-independent constraints and synthesis script, pad insertion and prompts for technology-specific pads, and translation of technology-independent formulae into technology-dependent scripts. The configurable PCI bus interface is notable because it implements consistency checking of parameters, configuration-based installation, and automatic modification of HDL files.



Additionally, prior art synthesis techniques do choose different mappings based on user goal specifications, allowing the mapping to optimize for speed, power, area, or target components. On this point, in the prior art it is not possible to get feedback on the effect of reconfiguring the processor in these ways without taking the design through the entire mapping process. Such feedback could be used to direct further reconfiguration of the processor until the system design goals are achieved.

The second category of prior art work in the area of configurable processor generation, i.e., automatic retargetting of compilers and assemblers) encompasses a rich area of academic research; see, e.g., Hanono et al., "Instruction Selection, Resource Allocation and Scheduling in the AVIV Retargetable Code Generator" (representation of machine instructions used for automatic creation of code generators); Fauth et al., "Describing Instruction Set Processors Using nML"; Ramsey et al., "Machine Descriptions to Build Tools for Embedded Systems"; Aho et al, "Code Generation Using Tree Matching and Dynamic Programming" (algorithms to match up transformations associated with each machine instruction, e.g., add, load, store, branch, etc., with a sequence of program operations represented by some machine-independent intermediate form using methods such as pattern matching); and Cattell, "Formalization and Automatic Derivation of Code Generators" (abstract descriptions of machine architectures used for compiler research).

Once the processor has been designed, its operation must be verified. That is, processors generally execute instructions from a stored program using a pipeline with each stage suited to one phase of the instruction execution. Therefore, changing or adding an instruction or changing the configuration may require widespread changes in the processor's logic so each of the multiple pipeline stages can perform the appropriate action on each such instruction. Configuration of a processor requires that it be re-verified, and that this verification adapt to the changes and additions. This is not a simple task. Processors are complex logic devices with extensive internal data and control state, and the combinatorics of control and data and program make processor verification a demanding art. Adding to the difficulty of processor verification is the difficulty in developing appropriate verification tools. Since verification is not automated in prior art techniques, its flexibility, speed and reliability is less than optimal.

In addition, once the processor is designed and verified it is not particularly useful if it cannot be programmed easily. Processors are generally programmed with the aid of extensive software tools, including compilers, assemblers, linkers, debuggers, simulators and profilers. When the processor changes, the software tools must change as well. It does no good to add an instruction if that instruction cannot be compiled, assembled, simulated or debugged. The cost of software changes associated with processor modifications and enhancements has been a major impediment to flexible processor design in the prior art.

Thus, it is seen that prior art processor design is of a level of difficulty that processors generally are not typically designed or modified for a specific application. Also, it can be seen that considerable improvements in system efficiency are possible if processors could be configured or extended for specific applications. Further, the efficiency and effectiveness of the design process could be enhanced if it were

able to use feedback on implementation characteristics such as power consumption, speed, etc. in refining a processor design. Moreover, in the prior art once a processor is modified, a great deal of effort is required to verify the correct operation of the processor after modification. Finally, although prior art techniques provide for limited processor configurability, they fail to provide for the generation of software development tools tailored for use with the configured processor.

While a system meeting the above criteria would certainly be an improvement over the art, improvements could be made -- for example, there is a need for a processor system with instructions that access or modify information stored in special registers, i.e., processor state, which significantly restricts the range of instructions obtainable and therefore limit the amount of performance improvement achievable.

Also, inventing new application-specific instructions involves complicated tradeoffs between cycle count reduction, additional hardware resources and CPU cycle-time impact. Another challenge is to obtain efficient hardware implementations for the new instructions without involving applications developers in the often tricky details of high-performance microprocessor implementations.

The above system gives the user flexibility to design a processor well-suited for her application, but is cumbersome for interactive development of hardware and software. To more fully understand this problem, consider a typical approach used by many software designers to tune the performance of their software application. They will typically think of a potential improvement, modify their software to use that potential improvement, recompile their software source to generate a runnable application containing that potential improvement and then evaluate the potential improvement. Depending on the results of that evaluation, they might keep or discard the potential improvement. Typically, the entire process can be completed in only a few minutes. This allows the user to experiment freely, quickly trying out and keeping or discarding ideas. In some cases, just evaluating a potential idea is complicated. The user might want to test the idea in a large variety of situations. In such cases, the user often keeps multiple versions of the compiled application: one original version and another version containing the potential improvement. In some cases, potential improvements might interact, and the user might keep more than two copies of the application, each using a different subset of the potential improvements. By keeping multiple versions, the user can easily test the different versions repeatedly under different circumstances.

Users of configurable processors would like to interactively develop hardware and software jointly in a similar fashion to the way that software developers develop software on traditional processors. Consider the case of users adding custom instructions to a configurable processor. Users would like to interactively add potential instructions to their processor and test and evaluate those instructions on their particular application. With prior art systems this is difficult for three reasons.

First, after proposing a potential instruction, the user must wait an hour or more before obtaining a compiler and simulator that can take advantage of the instruction.



Second, when the user wishes to experiment with many potential instructions, the user must create and keep a software development system for each. The software development system can be very large. Keeping many versions can become unmanageable.

Finally, the software development system is configured for the entire processor. That makes it difficult to separate the development process among different engineers. Consider an example where two developers are working on a particular application. One developer might be responsible for deciding on cache characteristics of the processor and another responsible for adding customized instructions. While the work of the two developers is related, each piece is sufficiently separable so that each developer can work on her task in isolation. The cache developer might initially propose a particular configuration. The other developer starts with that configuration and tries out several instructions, building a software development system for each potential instruction. Now, the cache developer modifies the proposed cache configuration. The other developer must now rebuild every one of her configurations, since each of her configurations assumed the original cache configuration. With many developers working on a project, organizing the different configurations can quickly become unmanageable.

## BRIEF SUMMARY OF THE INVENTION

The present invention overcomes these problems of the prior art and has an object of providing a system which can automatically configure a processor by generating both a description of a hardware implementation of the processor and a set of software development tools for programming the processor from the same configuration specification.

It is another object of the present invention to provide such a system which can optimize the hardware implementation and the software tools for various performance criteria.

It is still another object of the present invention to provide such a system that permits various types of configurability for the processor, including extensibility, binary selection and parametric modification.

It is yet another object of the present invention to provide such a system which can describe the instruction set architecture of the processor in a language which can easily be implemented in hardware.

It is a further object of the present invention to provide a system and method for developing and implementing instruction set extensions which modify processor state.

It is another object of the present invention to provide a system and method for developing and implementing instruction set extensions that modify configurable processor registers.

It is still another object of the present invention to allow the user to customize a processor configuration by adding new instructions and within minutes, be able to evaluate that feature.

The above objects are achieved by providing an automated processor generation system which uses a description of customized processor instruction set options and extensions in a standardized language to develop a configured definition of a target instruction set, a Hardware Description Language description of circuitry necessary to implement the instruction set, and development tools such as a

compiler, assembler, debugger and simulator which can be used to generate software for the processor and to verify the processor. Implementation of the processor circuitry can be optimized for various criteria such as area, power consumption and speed. Once a processor configuration is developed, it can be tested and inputs to the system modified to iteratively optimize the processor implementation.

To develop an automated processor generation system according to the present invention, an instruction set architecture description language is defined and configurable processor/system configuration tools and development tools such as assemblers, linkers, compilers and debuggers are developed. This is part of the development process because although large portions of the tools are standard, they must be made to be automatically configured from the ISA description. This part of the design process is typically done by the designer or manufacturer of the automated processor design tool itself.

An automated processor generation system according to the present invention operates as follows. A user, e.g., a system designer, develops a configured instruction set architecture. That is, using the ISA definition and tools previously developed, a configurable instruction set architecture following certain ISA design goals is developed. Then, the development tools and simulator are configured for this instruction set architecture. Using the configured simulator, benchmarks are run to evaluate the effectiveness of the configurable instruction set architecture, and the core revised based on the evaluation results. Once the configurable instruction set architecture is in a satisfactory state, a verification suite is developed for it.

Along with these software aspects of the process, the system attends to hardware aspects by developing a configurable processor. Then, using system goals such as cost, performance, power and functionality and information on available processor fabs, the system designs an overall system architecture which takes configurable ISA options, extensions and processor feature selection into account. Using the overall system architecture, development software, simulator, configurable instruction set architecture and processor HDL implementation, the processor ISA, HDL implementation, software and simulator are configured by the system and system HDL is designed for system-on-a-chip designs. Also, based on the system architecture and specifications of chip foundries, a chip foundry is chosen based on an evaluation of foundry capabilities with respect to the system HDL (not related to processor selection as in the prior art). Finally, using the foundry's standard cell library, the configuration system synthesizes circuitry, places and routes it, and provides the ability to re-optimize the layout and timing. Then, circuit board layouts are designed if the design is not of the single-chip type, chips are fabricated, and the boards are assembled.

As can be seen above, several techniques are used to facilitate extensive automation of the processor design process. The first technique used to address these issues is to design and implement specific mechanisms that are not as flexible as an arbitrary modification or extension, but which nonetheless allow significant functionality improvements. By constraining the arbitrariness of the change, the problems associated with it are constrained.



The second technique is to provide a single description of the changes and automatically generate the modifications or extensions to all affected components. Processors designed with prior art techniques have not done this because it is often cheaper to do something once manually than to write a tool to do it automatically and use the tool once. The advantage of automation applies when the task is repeated many times.

A third technique employed is to build a database to assist in estimation and automatic configuration for subsequent user evaluation.

Finally, a fourth technique is to provide hardware and software in a form that lends itself to configuration. In an embodiment of the present invention some of the hardware and software are not written directly in standard hardware and software languages, but in languages enhanced by the addition of a preprocessor that allows queries of the configuration database and the generation of standard hardware and software language code with substitutions, conditionals, replication, and other modifications. The core processor design is then done with hooks that allow the enhancements to be linked in.

To illustrate these techniques, consider the addition of application-specific instructions. By constraining the method to instructions that have register and constant operands and which produce a register result, the operation of the instructions can be specified with only combinatorial (stateless, feedback free) logic. This input specifies the opcode assignments, instruction name, assembler syntax and the combinatorial logic for the instructions, from which tools generate:

-- instruction decode logic for the processor to recognize the new opcodes;

-- addition of a functional unit to perform the combinatorial logic function on register operands;

-- inputs to the instruction scheduling logic of the processor to make sure the instruction issues only when its operands are valid;

-- assembler modifications to accept the new opcode and its operands and generate the correct machine code;

-- compiler modifications to add new intrinsic functions to access the new instructions;

-- disassembler/debugger modifications to interpret the machine code as the new instruction;

-- simulator modifications to accept the new opcodes and to perform the specified logic function; and

-- diagnostic generators which generate both direct and random code sequences that contain and check the results of the added instructions.

All of the techniques above are employed to add application-specific instructions. The input is constrained to input and output operands and the logic to evaluate them. The changes are described in one place and all hardware and software modifications are derived from that description. This facility shows how a single input can be used to enhance multiple components.

The result of this process is a system that is much better at meeting its application needs than existing art because tradeoffs between the processor and the rest of the system logic can be made much later in the design process. It is superior to many of the prior art approaches discussed above in that its

configuration may be applied to many more forms of representation. A single source may be used for all ISA encoding, software tools and high-level simulation may be included in a configurable package, and flow may be designed for iteration to find an optimal combination of configuration values. Further, while previous methods focused only on hardware configuration or software configuration alone without a single user interface for control, or a measurement system for user-directed redefinition, the present invention contributes to complete flow for configuration of processor hardware and software, including feedback from hardware design results and software performance to aid selection of optimal configuration.

These objects are achieved according to an aspect of the present invention by providing an automated processor design tool which uses a description of customized processor instruction set extensions in a standardized language to develop a configurable definition of a target instruction set, a Hardware Description Language description of circuitry necessary to implement the instruction set, and development tools such as a compiler, assembler, debugger and simulator which can be used to develop applications for the processor and to verify it. The standardized language is capable of handling instruction set extensions which modify processor state or use configurable processors. By providing a constrained domain of extensions and optimizations, the process can be automated to a high degree, thereby facilitating fast and reliable development.

The above objects are yet further achieved according to another aspect of the present invention which provides a system in which the user is able to keep multiple sets of potential instructions or state (hereinafter the combination of potential configurable instructions or state will be referred to collectively as "processor enhancements") and easily switch between them when evaluating their application.

The user selects and builds a base processor configuration using the methods described herein. The user creates a new set of user-defined processor enhancements and places them in a file directory. The user then invokes a tool that processes the user enhancements and transforms them into a form usable by the base software development tools. This transformation is very quick since it involves only the user-defined enhancements and does not build an entire software system. The user then invokes the base software development tools, telling the tools to dynamically use the processor enhancements created in the new directory. Preferably, the location of the directory is given to the tools either via a command line option or via an environment variable. To further simplify the process, the user can use standard software makefiles. These enable the user to modify their processor instructions and then via a single make command, process the enhancements and use the base software development system to rebuild and evaluate their application in the context of the new processor enhancements.

The invention overcomes the three limitations of the prior art approach. Given a new set of potential enhancements, the user can evaluate the new enhancements in a matter of minutes. The user can keep many versions of potential enhancements by creating new directories for each set. Since the directory only contains descriptions of the new enhancements and not the entire software system, the storage space required is minimal. Finally, the new enhancements are decoupled from the rest of the





configuration. Once the user has created a directory with a potential set of new enhancements, she can use that directory with any base configuration.

BRIEF DESCRIPTION OF THE DRAWINGS

The above and other objects of the present invention will become readily apparent when reading the following detailed description taken in conjunction with the appended drawings in which:

FIGURE 1 is a block diagram of a processor implementing an instruction set according to a preferred embodiment of the present invention;

FIGURE 2 is a block diagram of a pipeline used in the processor according to the embodiment;

FIGURE 3 shows a configuration manager in a GUI according to the embodiment;

FIGURE 4 shows a configuration editor in the GUI according to the embodiment;

FIGURE 5 shows different types of configurability according to the embodiment;

FIGURE 6 is a block diagram showing the flow of processor configuration in the embodiment;

FIGURE 7 is a block diagram of an instruction set simulator according to the embodiment.

FIGURE 8 is a block diagram of an emulation board for use with a processor configured according to the present invention.

FIGURE 9 is a block diagram showing the logical architecture of a configurable processor according to the embodiment;

FIGURE 10 is a block diagram showing the addition of a multiplier to the architecture of FIG. 9;

FIGURE 11 is a block diagram showing the addition of a multiply-accumulate unit to the architecture of FIG. 9;

FIGURES 12 and 13 are diagrams showing the configuration of a memory in the embodiment; and

FIGURES 14 and 15 are diagrams showing the addition of user-defined functional units in the architecture of FIG. 8.

FIGURE 16 is a block diagram showing the flow of information between system components in another preferred embodiment;

FIGURE 17 is a block diagram showing how custom code is generated for the software development tools in the embodiment;

FIGURE 18 is a block diagram showing the generation of various software modules used in another preferred embodiment of the present invention;

FIGURE 19 is a block diagram of a pipeline structure in a configurable processor according to the embodiment;

FIGURE 20 is a state register implementation according to the embodiment;

FIGURE 21 is a diagram of additional logic needed to implement the state register implementation in the embodiment;

FIGURE 22 is a diagram showing the combination of the next-state output of a state from several semantic blocks and selection one to input to a state register according to the embodiment;

FIGURE 23 shows logic corresponding to semantic logic according to the embodiment; and

FIGURE 24 shows the logic for a bit of state when it is mapped to the a bit of a user register in the embodiment.

## DETAILED DESCRIPTION OF PRESENTLY PREFERRED EMBODIMENTS

Generally, the automated processor generation process begins with a configurable processor definition and user-specified modifications thereto, as well as a user-specified application to which the processor is to be configured. This information is used to generate a configured processor taking the user modifications into account and to generate software development tools, e.g., compiler, simulator, assembler and disassembler, etc., for it. Also, the application is recompiled using the new software development tools. The recompiled application is simulated using the simulator to generate a software profile describing the configured processor's performance running the application, and the configured processor is evaluated with respect to silicon chip area usage, power consumption, speed, etc. to generate a hardware profile characterizing the processor circuit implementation. The software and hardware profile are fed back and provided to the user to enable further iterative configuration so that the processor can be optimized for that particular application.

An automated processor generation system 10 according to a preferred embodiment of the present invention has four major components as shown in FIG. 1: a user configuration interface 20 through which a user wishing to design a processor enters her configurability and extensibility options and other design constraints; a suite of software development tools 30 which can be customized for a processor designed to the criteria chosen by the user; a parameterized, extensible description of a hardware implementation of the processor; and a build system 50 receiving input data from the user interface, generating a customized, synthesizable hardware description of the requested processor, and modifying the software development tools to accommodate the chosen design. Preferably, the build system 50 additionally generates diagnostic tools to verify the hardware and software designs and an estimator to estimate hardware and software characteristics.

"Hardware implementation description", as used herein and in the appended claims, means one or more descriptions which describe aspects of the physical implementation of a processor design and, alone or in conjunction with one or more other descriptions, facilitate production of chips according to that design. Thus, components of the hardware implementation description may be at varying levels of abstraction, from relatively high levels such as hardware description languages through netlists and microcoding to mask descriptions. In this embodiment, however, the primary components of the hardware implementation description are written in an HDL, netlists and scripts.



Further, HDL as used herein and in the appended claims is intended to refer to the general class of hardware description languages which are used to describe microarchitectures and the like, and it is not intended to refer to any particular example of such languages.

In this embodiment, the basis for processor configuration is the architecture 60 shown in FIG. 2. A number of elements of the architecture are basic features which cannot be directly modified by the user. These include the processor controls section 62, the align and decode section 64 (although parts of this section are based on the user-specified configuration), the ALU and address generation section 66, the branch logic and instruction fetch, 68 and the processor interface 70. Other units are part of the basic processor but are user-configurable. These include the interrupt control section 72, the data and instruction address watch sections 74 and 76, the window register file 78, the data and instruction cache and tags sections 80, the write buffers 82 and the timers 84. The remaining sections shown in FIG. 2 are optionally included by the user.

A central component of the processor configuration system 10 is the user configuration interface 20. This is a module which preferably presents the user with a graphical user interface (GUI) by which it is possible to select processor functionality including reconfiguration of compiler and regeneration of assembler, disassembler and instruction set simulator (ISS); and preparation of input for launching of full processor synthesis, placement and routing. It also allows the user to take advantage of the quick estimation of processor area, power consumption, cycle time, application performance and code size for further iteration and enhancement of the processor configuration. Preferably, the GUI also accesses a configuration database to get default values and do error checking on user input.

To use the automated processor generation system 10 according to this embodiment to design a processor 60, a user inputs design parameters into the user configuration interface 20. The automated processor generation system 10 may be a stand-alone system running on a computer system under the control of the user; however, it preferably runs primarily on a system under the control of the manufacturer of the automated processor generation system 10. User access may then be provided over a communication network. For example, the GUI may be provided using a web browser with data input screens written in HTML and Java. This has several advantages, such as maintaining confidentiality of any proprietary back-end software, simplifying maintenance and updating of the back end software, and the like. In this case, to access the GUI the user may first log on to the system 10 to prove his identity.

Once the user has access, the system displays a configuration manager screen 86 as shown in FIG. 3. The configuration manager 86 is a directory listing all of the configurations accessible by the user. The configuration manager 86 in FIG. 3 shows that the user has two configurations, "`just intr`" and "`high prio`", the first having already been built, i.e., finalized for production, and the second yet to be built. From this screen 86 the user may build a selected configuration, delete it, edit it, generate a report specifying which configuration and extension options have been chosen for that configuration, or create a new configuration. For those configurations which have been built, such as "just intr", a suite of software development tools 30 customized for it can be downloaded.

Creating a new configuration or editing an existing one brings up the configuration editor 88 shown in FIG. 4. The configuration editor 88 has an "Options" section menu on the left showing the various general aspects of the processor 60 which can be configured and extended. When an option section is selected, a screen with the configuration options for that section appears on the right, and these options can be set with pull-down menus, memo boxes, check boxes, radio buttons and the like as is known in the art. Although the user can select options and enter data at random, preferably data is entered into each sequentially, since there are logical dependencies between the sections; for example, to properly display options in the "Interrupts" section, the number of interrupts must have been chosen in the "ISA Options" section.

In this embodiment, the following configuration options are available for each section:

- Goals
  - Technology for Estimation
    - Target ASIC technology: .18, .25, .35 micron
    - Target operating condition: typical, worst-case
  - Implementation Goals
    - Target speed: arbitrary
    - Gate count: arbitrary
    - Target power: arbitrary
    - Goal prioritization: speed, area power; speed, power, area
- ISA Options
  - Numeric Options
    - MAC16 with 40-bit accumulator: yes, no
    - 16-bit multiplier: yes, no
  - Exception Options
    - Number of interrupts: 0-32
    - High priority interrupt levels: 0-14
    - Enable Debugging: yes, no
    - Number of Timers: 0-3
  - Other
    - Byte Ordering: little endian, big endian
    - Number of registers available for call windows: 32, 64
- Processor Cache & Memory
    - Processor interface read width (bits): 32, 64, 128
    - Write-buffer entries (address/value pairs): 4, 8, 16, 32
  - Processor Cache
    - Instruction/Data cache size (kB): 1, 2, 4, 8, 16
    - Instruction/Data cache line size (kB): 16, 32, 64
- Peripheral Components
  - Timers
    - Timer interrupt numbers
    - Timer interrupt levels
- Debugging Support
    - Number of instruction address breakpoint registers: 0-2
    - Number of data address breakpoint registers: 0-2
    - Debug interrupt level
    - Trace port: yes, no
    - On-chip debug module: yes, no
    - Full scan: yes. no
- Interrupts
    - Source: external, software
    - Priority level
- System Memory Addresses
    - Vector and address calculation method: XTOS, manual
  - Configuration Parameters
    - RAM size, start address: arbitrary
    - ROM size, start address: arbitrary
    - XTOS: arbitrary
  - Configuration Specific Addresses
    - User exception vector: arbitrary
    - Kernel Exception vector: arbitrary
    - Register window over/underflow vector base: arbitrary
    - Reset vector: arbitrary
    - XTOS start address: arbitrary
    - Application start address: arbitrary
- TIE Instructions
    - (define ISA extensions)
- Target CAD Environment
  - Simulation
    - Verilog™: yes, no
  - Synthesis
    - Design Compiler™: yes, no
  - Place & Route
    - Apollo™: yes, no

Additionally, the system 10 may provide options for adding other functional units such as a 32-bit integer multiply/divide unit or a floating point arithmetic unit; a memory management unit; on-chip RAM and ROM options; cache associativity; enhanced DSP and coprocessor instruction set; a write-back cache; multiprocessor synchronization; compiler-directed speculation; and support for additional CAD packages. Whatever configuration options are available for a given configurable processor, they are preferably listed in a definition file (such as the one shown in Appendix A) which the system 10 uses for syntax checking and the like once the user has selected appropriate options.

From the above, one can see that the automated processor configuration system 10 provides two broad types of configurability 300 to the user as shown in FIG. 5: extensibility 302, which permits the user to define arbitrary functions and structures from scratch, and modifiability 304, which permits the user to select from a predetermined, constrained set of options. Within modifiability the system permits binary selection 306 of certain features, e.g., whether a MAC16 or a DSP should be added to the processor 60) and parametric specification 308 of other processor features, e.g., number of interrupts and cache size.

Many of the above configuration options will be familiar to those in the art; however, others merit particular attention. For example, the RAM and ROM options allow the designer to include scratch pad or firmware on the processor 10 itself. The processor 10 can fetch instructions or read and write data from these memories. The size and placement of the memories is configurable. In this embodiment, each of these memories is accessed as an additional set in a set-associative cache. A hit in the memory can be detected by comparison with a single tag entry.

The system 10 provides separate configuration options for the interrupt (implementing level 1 interrupts) and the high-priority interrupt option (implementing level 2-15 interrupts and non-maskable

interrupts) because each high-priority interrupt level requires three special registers, and these are thus more expensive.

The MAC16 with 40-bit accumulator option (shown at 90 in FIG. 2) adds a 16-bit multiplier/add function with a 40-bit accumulator, eight 16-bit operand registers and a set of compound instructions that combine multiply, accumulate, operand load and address update instructions. The operand registers can be loaded with pairs of 16-bit values from memory in parallel with multiply/accumulate operations. This unit can sustain algorithms with two loads and a multiply/accumulate per cycle.

The on-chip debug module (shown at 92 in FIG. 2) is used to access the internal, software-visible state of the processor 60 through the JTAG port 94. The on-chip debug module 92 provides support for exception generation to put the processor 60 in the debug mode; access to all program-visible registers or memory locations; execution of any instruction that the processor 60 is configured to execute; modification of the PC to jump to a desired location in the code; and a utility to allow return to a normal operation mode, triggered from outside the processor 60 via the JTAG port 94.

Once the processor 10 enters debug mode, it waits for an indication from the outside world that a valid instruction has been scanned in via the JTAG port 94. The processor then executes this instruction and waits for the next valid instruction. Once the hardware implementation of the processor 10 has been manufactured, this on-chip debug module 92 can be used to debug the system. Execution of the processor 10 can be controlled via a debugger running on a remote host. The debugger interfaces with the processor via the JTAG port 94 and uses the capability of the on-chip debug module 92 to determine and control the state of the processor 10 as well as to control execution of the instructions.

Up to three 32-bit counter/timers 84 may be configured. This entails the use of a 32-bit register which increments each clock cycle, as well as (for each configured timer) a compare register and a comparator which compares the compare register contents with the current clocked register count, for use with interrupts and similar features. The counter/timers can be configured as edge-triggered and can generate normal or high-priority internal interrupts.

The speculation option provides greater compiler scheduling flexibility by allowing loads to be speculatively moved to control flows where they would not always be executed. Because loads may cause exceptions, such load movement could introduce exceptions into a valid program that would not have occurred in the original. Speculative loads prevent these exceptions from occurring when the load is executed, but provide an exception when the data is required. Instead of causing an exception for a load error, speculative loads reset the valid bit of the destination register (new processor state associated with this option).

Although the core processor 60 preferably has some basic pipeline synchronization capability, when multiple processors are used in a system, some sort of communication and synchronization between processors is required. In some cases self-synchronizing communication techniques such as input and output queues are used. In other cases, a shared memory model is used for communication and it is necessary to provide instruction set support for synchronization because shared memory does not provide



the required semantics. For example, additional load and store instructions with acquire and release semantics can be added. These are useful for controlling the ordering of memory references in multiprocessor systems where different memory locations may be used for synchronization and data so that precise ordering between synchronization references must be maintained. Other instructions may be used to create semaphore systems known in the art.

In some cases, a shared memory model is used for communication, and it is necessary to provide instruction set support for synchronization because shared memory does not provide the required semantics. This is done by the multiprocessor synchronization option.

Perhaps most significantly among the configuration options are the TIE instruction definitions from which the designer-defined instruction execution unit 96 is built. The TIE™ (Tensilica Instruction Set Extensions) language developed by Tensilica Corporation of Santa Clara, California allows the user to describe custom functions for his applications in the form of extensions and new instructions to augment the base ISA. Additionally, due to TIE's flexibility it may be used to describe portions of the ISA which cannot be changed by the user; in this way, the entire ISA can be used to generate the software development tools 30 and hardware implementation description 40 uniformly. A TIE description uses a number of building blocks to delineate the attributes of new instructions as follows:

- instruction fields
- instruction classes
- instruction opcodes
- instruction semantics
- instruction operands
- constant tables

Instruction field statements `field` are used to improve the readability of the TIE code. Fields are subsets or concatenations of other fields that are grouped together and referenced by a name. The complete set of bits in an instruction is the highest-level superset field `inst`, and this field can be divided into smaller fields. For example,

```
field x    inst[11:8]
field y    inst[15:12]
field xy   {x, y}
```

defines two 4-bit fields, x and y, as sub-fields (bits 8-11 and 12-15, respectively) of a highest-level field `inst` and an 8-bit field xy as the concatenation of the x and y fields.

The statements opcode define opcodes for encoding specific fields. Instruction fields that are intended to specify operands, e.g., registers or immediate constants, to be used by the thus-defined opcodes, must first be defined with field statements and then defined with operand statements.

For example,

```
opcode    acs   op2 = 4'b0000    CUST0
opcode    adsel op2 = 4'b0001    CUST0
```

defines two new opcodes, `acs` and `adsel`, based on the previously-defined opcode CUST0 (`4'b0000` denotes a four bit-long binary constant 0000). The TIE specification of the preferred core ISA has the statements

```
field op0   inst[3:0]
field op1   inst[19:16]
field op2   inst[23:20]
opcode      QRST  op0 = 4'b0000
opcode      CUST0 op1=4'b0100 QRST
```

as part of its base definitions. Thus, the definitions of acs and adsel cause the TIE compiler to generate instruction decoding logic respectively represented by the following:

```
inst[23:0] = 0000 0110 xxxx xxxx xxxx 0000
inst[23:0] = 0001 0110 xxxx xxxx xxxx 0000
```

Instruction operand statements operand identify registers and immediate constants. Before defining a field as an operand, however, it must have been previously been defined as a field as above. If the operand is an immediate constant, the value of the constant can be generated from the operand, or it can be taken from a previously defined constant table defined as described below. For example, to encode an immediate operand the TIE code

```
field        offset     inst[23:6]
operand      offests4   offset {
     assign  offsets4 = {{14{offset[17]}}, offset}<<2;
}{
     wire        [31:0] t;
   assign  t = offsets4>>2;
   assign  offset = t[17:0];
}
```

defines an 18-bit field named offset which holds a signed number and an operand offsets4 which is four times the number stored in the offset field. The last part of the operand statement actually describes the circuitry used to perform the computations in a subset of the Verilog™ HDL for describing combinatorial circuits, as will be apparent to those skilled in the art.

Here, the wire statement defines a set of logical wires named t thirty-two bits wide. The first assign statement after the wire statement specifies that the logical signals driving the logical wires are the offsets4 constant shifted to the right, and the second assign statement specifies that the lower eighteen bits of t are put into the offset field. The very first assign statement directly specifies the value of the offsets4 operand as a concatenation of offset and fourteen replications of its sign bit (bit 17) followed by a shift-left of two bits.

For a constant table operand, the TIE code

```
table prime 16 {
     2, 3, 5, 7, 9, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
53
}
operand         prime_s    s {
     assign     prime_s = prime[s];
} {
     assign s = prime_s == prime[0]    ?     4'b0000 :
                prime_s == prime[1]    ?     4'b0001 :
```



```
                prime_s == prime[2]    ?     4'b0010 :
                prime_s == prime[3]    ?     4'b0011 :
                prime_s == prime[4]    ?     4'b0100 :
                prime_s == prime[5]    ?     4'b0101 :
                prime_s == prime[6]    ?     4'b0110 :
                prime_s == prime[7]    ?     4'b0111 :
                prime_s == prime[8]    ?     4'b1000 :
                prime_s == prime[9]    ?     4'b1001 :
                prime_s == prime[10]   ?     4'b1010 :
                prime_s == prime[11]   ?     4'b1011 :
                prime_s == prime[12]   ?     4'b1100 :
                prime_s == prime[13]   ?     4'b1101 :
                prime_s == prime[14]   ?     4'b1110 :
                                             4'b1111;
}
```

makes use of the table statement to define an array prime of constants (the number following the table name being the number of elements in the table) and uses the operand s as an index into the table prime to encode a value for the operand prime_s (note the use of Verilog™ statements in defining the indexing).

The instruction class statement iclass associates opcodes with operands in a common format. All instructions defined in an iclass statement have the same format and operand usage. Before defining an instruction class, its components must be defined, first as fields and then as opcodes and operands. For example, building on the code used in the preceding example defining opcodes acs and adsel, the additional statements

```
operand    art     t     {assign art = AR[t];} {}
operand    ars     s     {assign ars = AR[s];} {}
operand    arr     r     {assign AR[r] = arr;} {}
```

use the operand statement to define three register operands art, ars and arr (again note the use of Verilog™ statements in the definition). Then, the iclass statement

```
iclass     viterbi     {adsel, acs} {out arr, in art, in ars}
```

specifies that the operands adsel and acs belong to a common class of instructions viterbi which take two register operands art and ars as input and writes output to a register operand arr.

The instruction semantic statement semantic describes the behavior of one or more instructions using the same subset of Verilog™ used for coding operands. By defining multiple instructions in a single semantic statement, some common expressions can be shared and the hardware implementation can be made more efficient. The variables allowed in semantic statements are operands for opcodes defined in the statement's opcode list, and a single-bit variable for each opcode specified in the opcode list. This variable has the same name as the opcode and evaluates to 1 when the opcode is detected. It is used in the computation section (the Verilog™ subset section) to indicate the presence of the corresponding instruction.

For example, TIE code defining a new instruction ADD8_4 which performs additions of four 8-bit operands in a 32-bit word with respective 8-bit operands in another 32-bit word and a new instruction MIN16_2 which performs minimum selections between two 16-bit operands in a 32-bit word and respective 16-bit operands in another 32-bit word might read:

```
opcode ADD8_4  op2=4'b0000 CUST0
opcode MIN16_2 op2=4'b0001 CUST0

iclass add_min {ADD8_4, MIN16_2} {out arr, in ars, in art}

semantic add_min {ADD8_4, MIN16_2} {
      wire [31:0] add, min;
      wire [7:0] add3, add2, add1, add0;
      wire [15:0] min1, min0;
      assign add3 = art[31:24] + ars[31:24];
      assign add2 = art[23:16] + ars[23:16];
      assign add1 = art[15:8] + ars[15:8];
      assign add0 = art[7:0] + ars[7:0];
      assign add = {add3, add2, add1, add0};
      assign min1 = art[31:16] < ars[31:16] ? art[31:16] :
ars[31:16];
      assign min0 = art[15:0] < ars[15:0] ? art[15:0] : ars[15:0];
      assign min = {min1, min0};
      assign arr = (({32{{ADD8_4}}}) & (add)) | (({32{{MIN16_2}}})
& (min));
}
```

Here, op2, CUST0, arr, art and ars are predefined operands as noted above, and the opcode and iclass statements function as described above.

The semantic statement specifies the computations performed by the new instructions. As will be readily apparent to those skilled in the art, the second line within the semantic statement specifies the computations performed by the new ADD8_4 instruction, the third and fourth lines therein specify the computations performed by the new MIN16_2 instruction, and the last line within the section specifies the result written to the arr register.

Returning to the discussion of the user input interface, once the user has entered all of the configuration and extension options she desires, the build system 50 takes over. As shown in FIG. 6, the build system 50 receives a configuration specification constituted by the parameters set by the user and extensible features designed by the user, and combines them with additional parameters defining the core processor architecture, e.g., features not modifiable by the user, to create a single configuration specification 100 describing the entire processor. For example, in addition to the configuration settings 102 chosen by the user, the build system 50 might add parameters specifying the number of physical address bits for the processor's physical address space, the location of the first instruction to be executed by the processor 60 after reset, and the like.

The Xtensa™ Instruction Set Architecture (ISA) Reference Manual, Revision 1.0 by Tensilica, Inc. is incorporated herein by reference for the purposes of illustrating examples of instructions that can



be implemented within the configurable processor as core instructions and instructions which are available via the selection of configuration options.

The configuration specification 100 also includes an ISA package containing TIE language statements specifying the base ISA, any additional packages which might have been selected by the user such as a coprocessor package 98 (see FIG. 2) or a DSP package, and any TIE extensions supplied by the user. Additionally, the configuration specification 100 may have a number of statements setting flags indicative of whether certain structural features are to be included in the processor 60. For example,

```
IsaUseDebug                    1
IsaUseInterrupt                1
IsaUseHighPriorityInterrupt    0
IsaUseException                1
```

indicates that the processor will include the on-chip debug module 92, interrupt facilities 72 and exception handling, but not high-priority interrupt facilities.

Using the configuration specification 100, the following can be automatically generated as will be shown below:

-- instruction decode logic of the processor 60;

-- illegal instruction detection logic for the processor 60;

-- the ISA-specific portion of the assembler 110;

-- the ISA-specific support routines for the compiler 108;

-- the ISA-specific portion of the disassembler 100 (used by the debugger); and

-- the ISA-specific portion of the simulator 112.

It is valuable to generate these things automatically because an important configuration capability is to specify the inclusion of packages of instructions. For some things, it would be possible to implement this with conditionalized code in each of the tools to handle the instruction if it has been configured, but this is awkward; more importantly, it does not allow the system designer to easily add instructions for his system.

In addition to taking a configuration specification 100 as an input from the designer, it is also possible to accept goals and have the build system 50 determine the configuration automatically. The designer can specify goals for the processor 60. For example, clock rate, area, cost, typical power consumption, and maximum power consumption might be goals. Since some of the goals conflict (e.g., often performance can be increased only by increasing area or power consumption or both), the build system 50 also takes a priority ordering for the goals. The build system 50 then consults a search engine 106 to determine the set of configuration options available and determines how to set each option from an algorithm that attempts to simultaneously achieve the input goals.

The search engine 106 includes a database that has entries that describe the effect on the various metrics. Entries can specify that a particular configuration setting has an additive, multiplicative, or limiting effect on a metric. Entries can also be marked as requiring other configuration options as

prerequisites, or as being incompatible with other options. For example, the simple branch prediction option can specify a multiplicative or additive effect on Cycles Per Instruction (CPI -- a determinant of performance); a limit on clock rate, an additive effect on area, and an additive effect on power. It can be marked as incompatible with a fancier branch predictor, and dependent on setting the instruction fetch queue size to at least two entries. The value of these effects may be a function of a parameter, such as branch prediction table size. In general, the database entries are represented by functions that can be evaluated.

Various algorithms are possible for finding configuration settings that come closest to achieving the input goals. For example, a simple knapsack-packing algorithm considers each option in sorted order of value divided by cost and accepts any option specification that increases value while keeping cost below a specified limit. So, for example, to maximize performance while keeping power below a specified value, the options would be sorted by performance divided by power and each option that increases performance that can be configured without exceeding the power limit is accepted. More sophisticated knapsack algorithms provide some amount of backtracking.

A very different sort of algorithm for determining the configuration from goals and the design database is based on simulated annealing. A random initial set of parameters is used as the starting point, and then changes of individual parameters are accepted or rejected by evaluating a global utility function. Improvements in the utility function are always accepted while negative changes are accepted probabilistically based on a threshold that declines as the optimization proceeds. In this system the utility function is constructed from the input goals. For example, given the goals Performance > 200, Power < 100, Area < 4, with the priority of Power, Area, and Performance, the following utility function could be used:

```
Max((1-Power/100) * 0.5, 0) + (max((1-Area/4) * 0.3, 0) *
(if Power < 100 then 1 else (1 - Power/100)**2)) +
(max(Performance/200 * 0.2, 0) * (if Power < 100 then 1 else
(1-Power/100)**2)) * (if Area < 4 then 1 else (1 - area/4)**2))
```

which rewards decreases in power consumption until it is below 100 and then is neutral, rewards decreases in area until it is below 4, and then is neutral, and rewards increases in performance until it is above 200, and then is neutral. There are also components that reduce the area usage when power is out of spec and that reduce the performance usage when power or area are out of spec.

Both these algorithms and others can be used to search for configurations that satisfy the specified goals. What is important is that the configurable processor design has been described in a design database that has prerequisite and incompatibility option specifications and the impact of the configuration options on various metrics.

The examples we have given have used hardware goals that are general and not dependent on the particular algorithm being run on the processor 60. The algorithms described can also be used to select configurations well suited for specific user programs. For example, the user program can be run with a



cache accurate simulator to measure the number of cache misses for different types of caches with different characteristics such as different sizes, different line sizes and different set associativities. The results of these simulations can be added to the database used by the search algorithms of search engine 106 described to help select the hardware implementation description 40.

Similarly, the user algorithm can be profiled for the presence of certain instructions that can be optionally implemented in hardware. For example, if the user algorithm spends a significant time doing multiplications, the search engine 106 might automatically suggest including a hardware multiplier. Such algorithms need not be limited to considering one user algorithm. The user can feed a set of algorithms into the system, and the search engine 106 can select a configuration that is useful on average to the set of user programs.

In addition to selecting preconfigured characteristics of the processors 60, the search algorithms can also be used to automatically select or suggest to the users possible TIE extensions. Given the input goals and given examples of user programs written perhaps in the C programming language, these algorithms would suggest potential TIE extensions. For TIE extensions without state, compiler-like tools can be embodied with pattern matchers. These pattern matchers walk expression nodes in a bottom up fashion searching for multiple instruction patterns that could be replaced with a single instruction. For example, say that the user C program contains the following statements.

```
x = (y+z) << 2;
x2 = (y2+z2) << 2;
```

The pattern matcher would discover that the user in two different locations adds two numbers and shifts the result two bits to the left. The system would add to a database the possibility of generating a TIE instruction that adds two numbers and shifts the result two bits to the left.

The build system 50 keeps track of many possible TIE instructions along with a count of how many times they appear. Using a profiling tool, the system 50 also keeps track of how often each instruction is executed during the total execution of the algorithm. Using a hardware estimator, the system 50 keeps track of how expensive in hardware it would be to implement each potential TIE instruction. These numbers are fed into the search heuristic algorithm to select a set of potential TIE instructions that maximize the input goals; goals such as performance, code size, hardware complexity and the like.

Similar but more powerful algorithms are used to discover potential TIE instructions with state. Several different algorithms are used to detect different types of opportunities. One algorithm uses a compiler-like tool to scan the user program and detect if the user program requires more registers than are available on the hardware. As known to practitioners in the art, this can be detected by counting the number of register spills and restores in the compiled version of the user code. The compiler-like tool suggests to the search engine a coprocessor with additional hardware registers 98 but supporting only the operations used in the portions of the user's code that has many spills and restores. The tool is responsible for informing the database used by the search engine 106 of an estimate of the hardware cost of the

coprocessor as well as an estimate of how the user's algorithm performance is improved. The search engine 106, as described before, makes a global decision of whether or not the suggested coprocessor 98 leads to a better configuration.

Alternatively or in conjunction therewith, a compiler-like tool checks if the user program uses bit-mask operations to insure that certain variables are never larger than certain limits. In this situation, the tool suggests to the search engine 106 a co-processor 98 using data types conforming to the user limits (for example, 12 bit or 20 bit or any other size integers). In a third algorithm used in another embodiment, used for user programs in C++, a compiler-like tool discovers that much time is spent operating on user defined abstract data types. If all the operations on the data type are suitable for TIE, the algorithm proposes to the search engine 106 implementing all the operations on the data type with a TIE coprocessor.

To generate the instruction decode logic of the processor 60, one signal is generated for each opcode defined in the configuration specification. The code is generated by simply rewriting the

```
opcode NAME FIELD = VALUE
```

declaration to the HDL statement

```
assign NAME = FIELD == VALUE;
```

and the

```
opcode NAME FIELD = VALUE PARENTNAME [FIELD2 = VALUE2]
```

to

```
assign NAME = PARENTNAME & (FIELD == VALUE)
```

The generation of register interlock and pipeline stall signals has also been automated. This logic is also generated based on the information in the configuration specification. Based on register usage information contained in the `iclass` statement and the latency of the instruction the generated logic inserts a stall (or bubble) when the source operand of the current instruction depends on the destination operand of a previous instruction which has not completed. The mechanism for implementing this stall functionality is implemented as part of the core hardware.

The illegal instruction detection logic is generated by NOR'ing together the individual generated instruction signals AND'ed with their field restrictions:

```
assign illegalinst = !(INST1 | INST2… | INSTn);
```

The instruction decode signals and the illegal instruction signal are available as outputs of the decode module and as inputs to the hand-written processor logic.

To generate other processor features, this embodiment uses a Verilog™ description of the configurable processor 60 enhanced with a Perl-based preprocessor language. Perl is a full-featured language including complex control structures, subroutines, and I/O facilities. The preprocessor, which in an embodiment of the present invention is called TPP (as shown in the source listing in Appendix B, TPP is itself a Perl program), scans its input, identifies certain lines as preprocessor code (those prefixed by a



semicolon for TPP) written in the preprocessor language (Perl for TPP), and constructs a program consisting of the extracted lines and statements to generate the text of the other lines. The non-preprocessor lines may have embedded expressions in whose place expressions generated as a result of the TPP processing are substituted. The resultant program is then executed to produce the source code, i.e., Verilog™ code for describing the detailed processor logic (as will be seen below, TPP is also used to configure the software development tools 30).

When used in this context, TPP is a powerful preprocessing language because it permits the inclusion of constructs such as configuration specification queries, conditional expressions and iterative structures in the Verilog™ code, as well as implementing embedded expressions dependent on the configuration specification 100 in the Verilog™ code as noted above. For example, a TPP assignment based on a database query might look like

```
; $endian = config_get_value("IsaMemoryOrder")
```

where `config_get_value` is the TPP function used to query the configuration specification 100, `IsaMemoryOrder` is a flag set in the configuration specification 100, and `$endian` is a TPP variable to be used later in generating the Verilog™ code.

A TPP conditional expression might be

```
; if (config_get_value("IsaMemoryOrder") eq "LittleEndian")
;       {do Verilog™ code for little endian ordering}
; else
;       {do Verilog™ code for big endian ordering}
```

Iterative loops can be implemented by TPP constructs such as

```
; for ($i=0; $i<$ninterrupts; $i++)
;       {do Verilog™ code for each of 1..N interrupts}
```

where `$i` is a TPP loop index variable and `$ninterrupts` is the number of interrupts specified for the processor 60 (obtained from the configuration specification 100 using `config_get_value`).

Finally, TPP code can be embedded into Verilog™ expressions such as

```
wire [`$ninterrupts-1`:0] srInterruptEn;
xtscenflop    #(`$ninterrupts`)    srintrenreg    (srInterruptEn,
srDataIn_W[`$ninterrupts-1`:0], srIntrEnWEn,!cReset,CLK);
```

where:

`$ninterrupts` defines the number of interrupts and determines the width (in terms of bits) of the `xtscenflop` module (a flip-flop primitive module);

`srInterruptEn` is the output of the flip-flop, defined to be a wire of appropriate number of bits;

`srDataIn_W` is the input to the flip-flop, but only relevant bits are input based on number of interrupts;

`srIntrEnWEn` is the write enable of the flip-flop;

cReset is the clear input to the flip-flop ; and

CLK is the input clock to the flip-flop.

For example, given the following input to TPP:

```
;                    #                 Timer                 Interrupt
;                   if              ($IsaUseTimer)                {
wire                        [`$width-1`:0]                   srCCount;
wire                                                        ccountWEn;
//--------------------------------------------------------------------
//                           CCOUNT                          Register
//--------------------------------------------------------------------
assign  ccountWEn  =  srWEn_W  &&  (srWrAdr_W  ==  `SRCCOUNT);
xtflop #(`$width`) srccntreg (srCCount,(ccountWEn ? srDataIn_W :
srCCount+1),CLK);
;       for       ($i=0;      $i<$TimerNumber;      $i++)       (
//--------------------------------------------------------------------
//                          CCOMPARE                         Register
//--------------------------------------------------------------------
wire                   [`$width-1`:0]                 srCCompare`$i`;
wire                                                   ccompWEn`$i`;
assign ccompWEn`$i` = srWEn_W && (srWrAdr_W == `SRCCOMPARE`$i`);
xtenflop                  #(`$width`)                  srccmp`$i`reg
(srCCompare`$i`,srDataIn_W,ccompWEn`$i`,CLK);
assign   setCCompIntr`$i`   =   (srCCompare`$i`   ==   srCCount);
assign         clrCCompIntr`$i`          =             ccompWEn`$i`;
;                                                                 }
; } ## IsaUseTimer
```

and the declarations

```
$IsaUseTimer                        =                            1
$TimerNumber                        =                            2
$width = 32
```

TPP generates

```
wire                        [31:0]                         srCCount;
wire                                                      ccountWEn;
//--------------------------------------------------------------------
//                           CCOUNT                          Register
//--------------------------------------------------------------------
assign   ccountWEn  =  srWEn_W  &&  (srWrAdr_W  ==  `SRCCOUNT);
xtflop  #(32)  srccntreg  (srCCount,(ccountWEn  ?  srDataIn_W  :
srCCount+1),CLK);
//--------------------------------------------------------------------
//                          CCOMPARE                         Register
//--------------------------------------------------------------------
wire                        [31:0]                      srCCompare0;
wire                                                      ccompWEn0;

assign  ccompWEn0  =  srWEn_W  &&  (srWrAdr_W  ==  `SRCCOMPARE0);
xtenflop #(32) srccmp0reg (srCCompare0,srDataIn_W,ccompWEn0,CLK);
assign    setCCompIntr0    =    (srCCompare0    ==    srCCount);
```



```
assign           clrCCompIntr0           =            ccompWEn0;

//--------------------------------------------------------------------
//                          CCOMPARE                         Register
//--------------------------------------------------------------------
wire                        [31:0]                      srCCompare1;
wire                                                      ccompWEn1;

assign  ccompWEn1  =  srWEn_W  &&  (srWrAdr_W  ==  `SRCCOMPARE1);
xtenflop #(32) srccmp1reg (srCCompare1,srDataIn_W,ccompWEn1,CLK);
assign    setCCompIntr1    =    (srCCompare1    ==    srCCount);
assign clrCCompIntr1 = ccompWEn1;
```

The HDL description 114 thus generated is used to synthesize hardware for processor implementation using, e.g., the DesignCompiler™ manufactured by Synopsys Corporation in block 122. The result is then placed and routed using, e.g., Silicon Ensemble™ by Cadence Corporation or Apollo™ by Avant! Corporation in block 128. Once the components have been routed, the result can be used for wire back-annotation and timing verification in block 132 using, e.g., PrimeTime™ by Synopsys. The product of this process is a hardware profile 134 which can be used by the user to provide further input to the configuration capture routine for further configuration iterations.

As mentioned in connection with the logic synthesis section 122, one of the outcomes of configuring the processor 60 is a set of customized HDL files from which specific gate-level implementation can be obtained by using any of a number of commercial synthesis tools. One such a tool is Design Compiler™ from Synopsys. To ensure correct and high performance gate-level implementation, this embodiment provides scripts necessary to automate the synthesis process in the customer environment. The challenge in providing such scripts is to support a wide variety of synthesis methodologies and different implementation objectives of users. To address the first challenge, this embodiment breaks the scripts into smaller and functionally complete scripts. One such example is to provide a read script that can read all HDL files relevant to the particular processor configuration 60, a timing constraint script to set the unique timing requirement in the processor 60, and a script to write out synthesis results in a way that can be used for the placement and routing of the gate-level netlist. To address the second challenge, this embodiment provides a script for each implementation objective. One such example is to provide a script for achieving fastest cycle time, a script for achieving minimum silicon area, and a script for achieving minimum power consumption.

Scripts are used in other phases of processor configuration as well. For example, once the HDL model of the processor 60 has been written, a simulator can be used to verify the correct operation of the processor 60 as described above in connection with block 132. This is often accomplished by running many test programs, or diagnostics, on the simulated processor 60. Running a test program on the simulated processor 60 can require many steps such as generating an executable image of the test program, generating a representation of this executable image which can be read by the simulator 112, creating a temporary place where the results of the simulation can be gathered for future analysis, analyzing the

results of the simulation, and so on. In the prior art this was done with a number of throw-away scripts. These scripts had some built-in knowledge of the simulation environment, such as which HDL files should be included, where those files could be found in the directory structure, which files are required for the test bench, and so on. In the current design the preferred mechanism is to write a script template which is configured by parameter substitution. The configuration mechanism also uses TPP to generate a list of the files that are required for simulation.

Furthermore, in the verification process of block 132 it is often necessary to write other scripts which allow designers to run a series of test programs. This is often used to run regression suites that give a designer confidence that a given change in the HDL model does not introduce new bugs. These regression scripts were also often throw-away as they had many built-in assumptions about files names, locations, etc. As described above for the creation of a run script for a single test program the regression script is written as a template. This template is configured by substituting parameters for actual values at configuration time.

The final step in the process of converting an RTL description to a hardware implementation is to use a place and route (P&R) software to convert the abstract netlist into a geometrical representation. The P&R software analyzes the connectivity of the netlist and decides upon the placement of the cells. It then tries to draw the connections between all the cells. The clock net usually deserves special attention and is routed as a last step. This process can be both helped by providing the tools with some information, such as which cells are expected to be close together (known as soft grouping), relative placement of cells, which nets are expected to have small propagation delays, and so on.

To make this process easier and to ensure that the desired performance goals are met -- cycle time, area, power dissipation -- the configuration mechanism produces a set of scripts or input files for the P&R software. These scripts contain information as described above such as relative placements for cells. The scripts also contain information such as how many supply and ground connections are required, how these should be distributed along the boundary, etc. The scripts are generated by querying a database that contains information on how many soft groups to create and what cells should be contained in them, which nets are timing critical, etc. These parameters change based on which options have been selected. These scripts must be configurable depending on the tools to be used to do the place and route.

Optionally the configuration mechanism can request more information from the user and pass it to the P&R scripts. For example the interface can ask the user the desired aspect ratio of the final layout, how many levels of buffering should be inserted in the clock tree, which side the input and output pins should be located on, relative, or absolute, placement of these pins, width and location of the power and ground straps, and so on. These parameters would then be passed on to the P&R scripts to generate the desired layout.

Even more sophisticated scripts can be used that allow for example a more sophisticated clock tree. One common optimization done to reduce power dissipation is to gate the clock signal. However, this makes clock tree synthesis a much harder problem since it is more difficult to balance the delay of all



branches. The configuration interface could ask the user for the correct cells to use for the clock tree and the perform part, or all, of the clock tree synthesis. It would do this by having some knowledge of where the gated clocks are located in the design and estimating the delay form the qualifying gate to the clock input of the flip-flops. It would than give a constraint to the clock tree synthesis tool to match the delay of the clock buffer with the delay of the gating cells. In the current implementation this is done by a general purpose Perl script. This script reads gated clock information produced by the configuration agent based on which options are selected. The Perl script is run once the design has been placed and routed but before final clock tree synthesis is done.

Further improvement can be made to the profile process described above. Specifically, we will describe a process by which the user can obtain the similar hardware profile information almost instantaneously without spending hours running those CAD tools. This process has several steps.

The first step in this process is to partition the set of all configuration options into groups of orthogonal options such that effect of an option in a group on the hardware profile is independent of options in any other group. For example, the impact of MAC16 unit to the hardware profile is independent of any other options. So, an option group with only the MAC16 option is formed. A more complicated example is an option group containing interrupt options, high-level interrupt options and timer options, since the impact on the hardware profile is determined by the particular combination of these options.

The second step is to characterize the hardware profile impact of each option groups. The characterization is done by obtaining hardware profile impact for various combinations of options in the group. For each combination, the profile is obtained using a previously-described process in which an actual implementation is derived and its hardware profile is measured. Such information is stored in an estimation database.

The last step is to derive specific formulae for computing hardware profile impact by particular combinations of options in the option groups using curve fitting and interpolation techniques. Depending on the nature of the options, different formulae are used. For example, since each additional interrupt vector adds about the same logic to the hardware, we use linear function to model its hardware impact. In another example, having a timer unit requires the high-priority interrupt option, so the formula for hardware impact of the timer option is conditional formulae involving several options.

It is useful to provide quick feedback on how architectural choices may affect the runtime performance and code size of applications. Several sets of benchmark programs from multiple application domains are chosen. For each domain, a database is prebuilt that estimates how different architectural design decisions will affect the runtime performance and code size of the applications in the domain. As the user varies the architectural design, the database is queried for the application domain that interests the user or for multiple domains. The results of the evaluation are presented to the user so she can get an estimate on the tradeoff between software benefits and hardware costs.

The quick evaluation system can be easily extended to provide the user with suggestions on how to modify a configuration to further optimize the processor. One such example is to associate each configuration option with a set of numbers representing the incremental impact of the option on various cost metrics such as area, delay and power. Computing the incremental cost impact for a given option is made easy with the quick evaluation system. It simply involves two calls to the evaluation system, with and without the option. The difference in the costs for the two evaluations represents the incremental impact of the option. For example, the incremental area impact of the MAC16 option is computed by evaluating the area cost of two configurations, with and without the MAC16 option. The difference is then displayed with the MAC16 option in the interactive configuration system. Such a system can guide the user toward an optimal solution through a series of single-step improvements.

Moving on to the software side of the automated processor configuration process, this embodiment of this invention configures software development tools 30 so that they are specific to the processor. The configuration process begins with software tools 30 that can be ported to a variety of different systems and instruction set architectures. Such retargetable tools have been widely studied and are well-known in the art. This embodiment uses the GNU family of tools, which is free software, including for example, the GNU C compiler, GNU assembler, GNU debugger, GNU linker, GNU profiler, and various utility programs. These tools 30 are then automatically configured by generating portions of the software directly from the ISA description and by using TPP to modify portions of the software that are written by hand.

The GNU C compiler is configured in several different ways. Given the core ISA description, much of the machine-dependent logic in the compiler can be written by hand. This portion of the compiler is common to all configurations of the configurable processor instruction set, and retargeting by hand allows fine-tuning for best results. However, even for this hand-coded portion of the compiler, some code is generated automatically from the ISA description. Specifically, the ISA description defines the sets of constant values that can be used in immediate fields of various instructions. For each immediate field, a predicate function is generated to test if a particular constant value can be encoded in the field. The compiler uses these predicate functions when generating code for the processor 60. Automating this aspect of the compiler configuration eliminates an opportunity for inconsistency between the ISA description and the compiler, and it enables changing the constants in the ISA with minimal effort.

Several aspects of the compiler are configured via preprocessing with TPP. For the configuration options controlled by parameter selection, corresponding parameters in the compiler are set via TPP. For example, the compiler has a flag variable to indicate whether the target processor 60 uses big endian or little endian byte ordering, and this variable is set automatically using a TPP command that reads the endianness parameter from the configuration specification 100. TPP is also used to conditionally enable or disable hand-coded portions of the compiler which generate code for optional ISA packages, based on whether the corresponding packages are enabled in the configuration specification 100. For example, the code to generate multiply/accumulate instructions is only included in the compiler if the configuration specification includes the MAC16 option 90.

The compiler is also configured to support designer-defined instructions specified via the TIE language. There are two levels of this support. At the lowest level, the designer-defined instructions are available as macros, intrinsic functions, or inline (extrinsic) functions in the code being compiled. This embodiment of this invention generates a C header file defining inline functions as "inline assembly" code (a standard feature of the GNU C compiler). Given the TIE specification of the designer-defined opcodes and their corresponding operands, generating this header file is a straightforward process of translating to the GNU C compiler's inline assembly syntax. An alternative implementation creates a header file containing C preprocessor macros that specify the inline assembly instructions. Yet another alternative uses TPP to add intrinsic functions directly into the compiler.

The second level of support for designer-defined instructions is provided by having the compiler automatically recognize opportunities for using the instructions. These TIE instructions could be directly defined by the user or created automatically during the configuration process. Prior to compiling the user application, the TIE code is automatically examined and converted into C equivalent functions. This is the same step used to allow fast simulation of TIE instructions. The C equivalent functions are partially compiled into a tree-based intermediate representation used by the compiler. The representation for each TIE instruction is stored in a database. When the user application is compiled, part of the compilation process is a pattern matcher. The user application is compiled into the tree-based intermediate representation. The pattern matcher walks bottom-up every tree in the user program. At each step of the walk, the pattern matcher checks if the intermediate representation rooted at the current point matches any of the TIE instructions in the database. If there is a match, the match is noted. After finishing to walk each tree, the set of maximally sized matches are selected. Each maximal match in the tree is replaced with the equivalent TIE instruction.

The algorithm described above will automatically recognize opportunities to use stateless TIE instructions. Additional approaches can also be used to automatically recognize opportunities to use TIE instructions with state. A previous section described algorithms for automatically selecting potential TIE instructions with state. The same algorithms are used to automatically use the TIE instructions in C or C++ applications. When a TIE coprocessor has been defined to have more registers but a limited set of operations, regions of code are scanned to see if they suffer from register spilling and if those regions only use the set of available operations. If such regions are found, the code in those regions is automatically changed to use the coprocessor 98 instructions and registers. Conversion operations are generated at the boundaries of the region to move the data in and out of the coprocessor 98. Similarly, if a TIE coprocessor has been defined to work on different size integers, regions of the code are examined to see if all data in the region is accessed as if it were the different size. For matching regions, the code is changed and glue code is added at the boundaries. Similarly if a TIE coprocessor 98 has been defined to

implement a C++ abstract data type, all the operations in that data type are replaced with the TIE coprocessor instructions.

Note that suggesting TIE instructions automatically and utilizing TIE instructions automatically are both useful independently. Suggested TIE instructions can also be manually used by the user via the intrinsic mechanism and utilizing algorithms can be applied to TIE instructions or coprocessors 98 designed manually.

Regardless of how designer-designed instructions are generated, either via inline functions or by automatic recognition, the compiler needs to know the potential side effects of the designer-defined instructions so that it can optimize and schedule these instructions. In order to improve performance, traditional compilers optimize user codes in order to maximize desired characteristics such as run-time performance, code size or power consumption. As is known to one well-versed in the art, such optimizations include things such as rearranging instructions or replacing certain instructions with other, semantically equivalent instructions. In order to perform optimizations well, the compiler must know how every instruction affects different portions of the machine. Two instructions that read and write different portions of the machine state can be freely reordered. Two instructions that access the same portion of the machine state can not always be reordered. For traditional processors, the state read and/or written by different instructions is hardwired, sometimes by table, into the compiler. In one embodiment of this invention, TIE instructions are conservatively assumed to read and write all the state of the processor 60. This allows the compiler to generate correct code but limits the ability of the compiler to optimize code in the presence of TIE instructions. In another embodiment of this invention, a tool automatically reads the TIE definition and for each TIE instruction discovers which state is read or written by said instruction. This tool then modifies the tables used by the compiler's optimizer to accurately model the effect of each TIE instruction.

Like the compiler, the machine-dependent portions of the assembler 110 include both automatically generated parts and hand-coded parts configured with TPP. Some of the features common to all configurations are supported with code written by hand. However, the primary task of the assembler 110 is to encode machine instructions, and instruction encoding and decoding software can be generated automatically from the ISA description.

Because instruction encoding and decoding are useful in several different software tools, this embodiment of this invention groups the software to perform those tasks into a separate software library. This library is generated automatically using the information in the ISA description. The library defines an enumeration of the opcodes, a function to efficiently map strings for opcode mnemonics onto members of the enumeration (`stringToOpcode`), and tables that for each opcode specify the instruction length (`instructionLength`), number of operands (`numberOfOperands`), operand fields, operand types (i.e., register or immediate) (`operandType`), binary encoding (`encodeOpcode`), and mnemonic string (`opcodeName`). For each operand field, the library provides accessor functions to encode



(`fieldSetFunction`) and decode (`fieldGetFunction`) the corresponding bits in the instruction word. All of this information is readily available in the ISA description; generating the library software is merely a matter of translating the information into executable C code. For example, the instruction encodings are recorded in a C array variable where each entry is the encoding for a particular instruction, produced by setting each opcode field to the value specified for that instruction in the ISA description; the `encodeOpcode` function simply returns the array value for a given opcode.

The library also provides a function to decode the opcode in a binary instruction `decodeInstruction`). This function is generated as a sequence of nested `switch` statements, where the outermost switch tests the subopcode field at the top of the opcode hierarchy, and the nested `switch` statements test the subopcode fields progressively lower in the opcode hierarchy. The generated code for this function thus has the same structure as the opcode hierarchy itself.

Given this library for encoding and decoding instructions, the assembler 110 is easily implemented. For example, the instruction encoding logic in the assembler is quite simple:

```
AssembleInstruction (String mnemonic, int arguments[])
begin
      opcode = stringToOpcode(mnemonic);
      if (opcode == UNDEFINED)
            Error("Unknown opcode");
      instruction = encodeOpcode(opcode);
      numArgs = numberOfOperands(opcode);
      for i = 0, numArgs-1 do
      begin
            setFun = fieldSetFunction(opcode, i);
            setFun(instruction, arguments[i]);
      end
      return instruction;
end
```

Implementing a disassembler 110, which translates binary instructions into a readable form closely resembling assembly code, is equally straightforward:

```
DisassembleInstruction (BinaryInstruction instruction)
begin
      opcode = decodeInstruction(instruction);
      instructionAddress += instructionLength(opcode);
      print opcodeName(opcode);
      // Loop through the operands, disassembling each
      numArgs = numberOfOperands(opcode);
      for i = 0, numArgs-1 do
      begin
            type = operandType(opcode, i);
            getFun = fieldGetFunction(opcode, i);
            value = getFun(opcode, i, instruction);
            if (i != 0) print ",";  // Comma separate operands
            // Print based on the type of the operand
            switch (type)
            case register:
                  print registerPrefix(type), value;
            case immediate:
                  print value;
```

```
            case pc_relative_label:
                 print instructionAddress + value;
            // etc. for more different operand types
      end
  end
```

This disassembler algorithm is used in a standalone disassembler tool and also in the debugger 130 to support debugging of machine code.

The linker is less sensitive to the configuration than the compiler and assembler 110. Much of the linker is standard and even the machine-dependent portions depend primarily on the core ISA description and can be hand-coded for a particular core ISA. Parameters such as endianness are set from the configuration specification 100 using TPP. The memory map of the target processor 60 is one other aspect of the configuration that is needed by the linker. As before, the parameters that specify the memory map are inserted into the linker using TPP. In this embodiment of the invention, the GNU linker is driven by a set of linker scripts, and it is these linker scripts that contain the memory map information. An advantage of this approach is that additional linker scripts can be generated later, without reconfiguring the processor 60 and without rebuilding the linker, if the memory map of the target system is different than the memory map specified when the processor 60 was configured. Thus, this embodiment includes a tool to configure new linker scripts with different memory map parameters.

The debugger 130 provides mechanisms to observe the state of a program as it runs, to single-step the execution one instruction at a time, to introduce breakpoints, and to perform other standard debugging tasks. The program being debugged can be run either on a hardware implementation of the configured processor or on the ISS 126. The debugger presents the same interface to the user in either case. When the program is run on a hardware implementation, a small monitor program is included on the target system to control the execution of the user's program and to communicate with the debugger via a serial port. When the program is run on the simulator 126, the simulator 126 itself performs those functions. The debugger 130 depends on the configuration in several ways. It is linked with the instruction encoding/decoding library described above to support disassembling machine code from within the debugger 130. The part of the debugger 130 that displays the processor's register state, and the parts of the debug monitor program and ISS 126 that provide that information to the debugger 130, are generated by scanning the ISA description to find which registers exist in the processor 60.

Other software development tools 30 are standard and need not be changed for each processor configuration. The profile viewer and various utility programs fall into this category. These tools may need to be retargeted once to operate on files in the binary format shared by all configurations of the processor 60, but they do not depend on either the ISA description or the other parameters in the configuration specification 100.

The configuration specification is also used to configure a simulator called the ISS 126 shown in FIG. 13. The ISS 126 is a software application that models the functional behavior of the configurable



processor instruction set. Unlike its counterpart processor hardware model simulators such as Synopsys VCS and Cadence Verilog XL and NC simulators, the ISS HDL model is an abstraction of the CPU during its instruction execution. The ISS 126 can run much faster than a hardware simulation because it does not need to model every signal transition for every gate and register in the complete processor design.

The ISS 126 allows programs generated for the configured processor 60 to be executed on a host computer. It accurately reproduces the processor's reset and interrupt behavior allowing low-level programs such as device drivers and initialization code to be developed. This is particularly useful when porting native code to an embedded application.

The ISS 126 can be used to identify potential problems such as architectural assumptions, memory ordering considerations and the like without needing to download the code to the actual embedded target.

In this embodiment, ISS semantics are expressed textually using a C-like language to build C operator building blocks that turn instructions into functions. For example, the rudimentary functionality of an interrupt, e.g., interrupt register, bit setting, interrupt level, vectors, etc., is modeled using this language.

The configurable ISS 126 is used for the following four purposes or goals as part of the system design and verification process:

-- debugging software applications before hardware becomes available;

-- debugging system software (e.g., compilers and operating system components);

-- comparing with HDL simulation for hardware design verification. ISS serves as a reference implementation of the ISA -- the ISS and processor HDL are both run for diagnostics and applications during processor design verification and traces from the two are compared; and

-- analyzing software application performance (this may be part of the configuration process, or it may be used for further application tuning after a processor configuration has been selected).

All the goals require that the ISS 126 be able to load and decode programs produced with the configurable assembler 110 and linker. They also require that ISS execution of instructions be semantically equivalent to the corresponding hardware execution and to the compiler's expectations. For these reasons, the ISS 126 derives its decode and execution behavior from the same ISA files used to define the hardware and system software.

For the first and last goals listed above, it is important for the ISS 126 to be as fast as possible for the required accuracy. The ISS 126 therefore permits dynamic control of the level of detail of the simulation. For example, cache details are not modeled unless requested, and cache modeling can be turned off and on dynamically. In addition, parts of the ISS 126 (e.g., cache and pipeline models) are configured before the ISS 126 is compiled so that the ISS 126 makes very few configuration-dependent choices of behavior at runtime. In this way, all ISS configurable behavior is derived from well-defined sources related to other parts of the system.

For the first and third goals listed above, it is important for the ISS 126 to provide operating system services to applications when these services are not yet available from the OS for the system under design (the target). It is also important for these services to be provided by the target OS when that is a relevant part of the debugging process. In this way the system provides a design for flexibly moving these services between ISS host and simulation target. The current design relies on a combination of ISS dynamic control (trapping SYSCALL instructions may be turned on and off) and the use of a special SIMCALL instruction to request host OS services.

The last goal requires the ISS 126 to model some aspects of processor and system behavior that are below the level specified by the ISA. In particular, the ISS cache models are constructed by generating C code for the models from Perl scripts which extract parameters from the configuration database 100. In addition, details of the pipeline behavior of instructions (e.g., interlocks based on register use and functional-unit availability requirements) are also derived from the configuration database 100. In the current implementation, a special pipeline description file specifies this information in a lisp-like syntax.

The third goal requires precise control of interrupt behavior. For this purpose, a special non-architectural register in the ISS 126 is used to suppress interrupt enables.

The ISS 126 provides several interfaces to support the different goals for its use:

-- a batch or command line mode (generally used in connection with the first and last goals);

-- a command loop mode, which provides non-symbolic debug capabilities, e.g. breakpoints, watchpoints, step, etc. -- frequently used for all four goals; and

-- a socket interface which allows the ISS 126 to be used by a software debugger as an execution backend (this must be configured to read and write the register state for the particular configuration selected).

-- a scriptable interface which allows very detailed debugging and performance analysis. In particular, this interface may be used to compare application behavior on different configurations. For example, at any breakpoint the state from a run on one configuration may be compared with or transferred to the state from a run on another configuration.

The simulator 126 also has both hand-coded and automatically generated portions. The hand-coded portions are conventional, except for the instruction decode and execution, which are created from tables generated from the ISA description language. The tables decode the instruction by starting from the primary opcode found in the instruction word to be executed, indexing into a table with the value of that field, and continuing until a leaf opcode, i.e., an opcode which is not defined in terms of other opcodes, is found. The tables then give a pointer to the code translated from the TIE code specified in the semantics declaration for the instruction. This code is executed to simulate the instruction.

The ISS 126 can optionally profile the execution of the program being simulated. This profiling uses a program counter sampling technique known in the art. At regular intervals, the simulator 126 samples the PC (program counter) of the processor being simulated. It builds a histogram with the



The ISS 126 can optionally profile the execution of the program being simulated. This profiling uses a program counter sampling technique known in the art. At regular intervals, the simulator 126 samples the PC (program counter) of the processor being simulated. It builds a histogram with the number of samples in each region of code. The simulator 126 also counts the number of times each edge in the call graph is executed by incrementing a counter whenever a call instruction is simulated. When the simulation is complete, the simulator 126 writes an output file containing both the histogram and call graph edge counts in a format that can be read by a standard profile viewer. Because the program 118 being simulated need not be modified with instrumentation code (as in standard profiling techniques), the profiling overhead does not affect the simulation results and the profiling is totally non-invasive.

It is preferable that the system make available hardware processor emulation as well as software processor emulation. For this purpose, this embodiment provides an emulation board. As shown in FIG. 6, the emulation board 200 uses a complex programmable logic device 202 such as the Altera Flex 10K200E to emulate, in hardware, a processor configuration 60. Once programmed with the processor netlist generated by the system, the CPLD device 202 is functionally equivalent to the final ASIC product. It provides the advantage that a physical implementation of the processor 60 is available that can run much faster than other simulation methods (like the ISS 126 or HDL) and is cycle accurate. However, it cannot reach the high frequency targets that the final ASIC device can get to.

This board enables the designer to evaluate various processor configuration options and start software development and debugging early in the design cycle. It can also be used for the functional verification of the processor configuration.

The emulation board 200 has several resources available on it to allow for easy software development, debugging and verification. These include the CPLD device 202 itself, EPROM 204, SRAM 206, synchronous SRAM 208, flash memory 210 and two RS232 serial channels 212. The serial channels 212 provide a communication link to UNIX or PC hosts for downloading and debugging user programs. The configuration of a processor 60, in terms of the CPLD netlist, is downloaded into the CPLD 202 through a dedicated serial link to device's configuration port 214 or through dedicated configuration ROMs 216.

The resources available on the board 200 are configurable to a degree as well. The memory map of the various memory elements on the board can be easily changed, because the mapping is done through a Programmable Logic Device (PLD) 217 which can be easily changed. Also, the caches 218 and 228 that the processor core uses are expandable by using larger memory devices and appropriately sizing the tag busses 222 and 224 that connect to the caches 218 and 228.

Using the board to emulate a particular processor configuration involves several steps. The first step is to obtain a set of RTL files which describe the particular configuration of the processor. The next step is to synthesize a gate-level netlist from the RTL description using any of a number of commercial synthesis tools. One such example is FPGA Express from Synopsys. The gate-level netlist can then be used to obtain a CPLD implementation using tools typically provided by vendors. One such tool is

Since one of the purposes of the emulation board is to support quick prototype implementation for debugging purposes, it is important that the CPLD implementation process outlined in the previous paragraph is automatic. To achieve this objective, the files delivered to users are customized by grouping all relevant files into a single directory. Then, a fully customized synthesis script is provided to be able to synthesize the particular processor configuration to the particular FPGA device selected by the customer. A fully customized implementation script to be used by the vendor tools is also generated. Such synthesis and implementation scripts guarantee functionally correct implementation with optimal performance. The functional correctness is achieved by including appropriate commands in the script to read in all RTL files relevant to the specific processor configuration by including appropriate commands to assign chip-pin locations based on I/O signals in the processor configuration and by including commands to obtain specific logic implementation for certain critical portions of the processor logic such at gated clocks. The script also improves the performance of the implementation by assigning detailed timing constraint to all processor I/O signals and by special processing of certain critical signals. One such example for timing constraints is assigning a specific input delay to a signal by taking into account the delay of that signal on the board. An example of critical signal treatment is to assign the clock signal to a dedicated global wire in order to achieve low clock skews on the CPLD chip.

Preferably, the system also configures a verification suite for the configured processor 60. Most verification of complex designs like microprocessors consists of a flow as follows:

-- build a test bench to stimulate the design and compare output either within the testbench or using an external model like the ISS 126;

-- write diagnostics to generate the stimulus;

-- measure coverage of verification using schemes like line coverage of finite state machine coverage HDL, declining bug rate, number of vectors run on the design; and

-- if the coverage is not sufficient -- write more diagnostics and maybe use tools to generate diagnostics to exercise the design further.

The present invention uses a flow that is somewhat similar, but all components of the flow are modified to account for the configurability of the design. This methodology consists of the following steps:

-- build a testbench for a particular configuration. Configuration of the testbench uses a similar approach as that described for the HDL and supports all options and extensions supported therein, i.e., cache sizes, bus interface, clocking, interrupt generation etc.;

-- run self-checking diagnostics on a particular configuration of the HDL. Diagnostics themselves are configurable to tailor them for a particular piece of hardware. The selection of which diagnostics to run is also dependent on the configuration;

-- run pseudo-randomly generated diagnostics and compare the processor state after the execution of each instruction against the ISS 126; and



-- measure of coverage of verification -- using coverage tools that measure functional as well as line coverage. Also, monitors and checkers are run along with the diagnostics to look for illegal states and conditions. All of these are configurable for a particular configuration specification.

All of the verification components are configurable. The configurability is implemented using TPP.

A test bench is a Verilog™ model of a system in which the configured processor 60 is placed. In the case of the present invention these test benches include:

-- caches, bus interface, external memory;

-- external interrupts and bus error generation; and

-- clock generation.

Since almost all of the above characteristics are configurable, the test bench itself needs to support configurability. So, for example, the cache size and width and number of external interrupts are automatically adjusted based on configuration.

The testbench provides stimulus to the device under test -- the processor 60. It does this by providing assembly level instructions (from diagnostics) that are preloaded into memory. It also generates signals that control the behavior of the processor 60 -- for example, interrupts. Also, the frequency and timing of these external signals is controllable and is automatically generated by the testbench.

There are two types of configurability for diagnostics. First, diagnostics use TPP to determine what to test. For example, a diagnostic has been written to test software interrupts. This diagnostic will need to know how many software interrupts there are in order to generate the right assembly code.

Second, the processor configuration system 10 must decide which diagnostics are suitable for this configuration. For example, a diagnostic written to test the MAC unit is not applicable to a processor 60 which does not include this unit. In this embodiment this is accomplished through the use of a database containing information about each diagnostic. The database may contain for each diagnostic the following information:

-- use the diagnostic if a certain option has been selected;

-- if the diagnostic cannot be run with interrupts;

-- if the diagnostic requires special libraries or handlers to run; and

-- if the diagnostic cannot be run with cosimulation with ISS 126.

Preferably the processor hardware description includes three types of test tools: test generator tools, monitors and coverage tools (or checkers), and a cosimulation mechanism. Test generation tools are tools that create a series of processor instructions in an intelligent fashion. They are sequences of pseudo-random test generators. This embodiment uses two types internally -- a specially-developed one called RTPG and another which is based on an external tool called VERA (VSG). Both have configurability built around them. Based on valid instructions for a configuration, they will generate a series of instructions. These tools will also be able to deal with newly defined instructions from TIE -- so

that these newly defined instructions are randomly generated for testing. This embodiment includes monitors and checkers that measure the coverage of the design verification.

Monitors and coverage tools are tools that are run alongside a regression run. Coverage tools monitor what the diagnostic is doing and the functions and logic of the HDL that it is exercising. All this information is collected throughout the regression run and is later analyzed to get some hints of what parts of the logic need further testing. This embodiment uses several functional coverage tools that are configurable. For example, for a particular finite state machine not all states are included depending on a configuration. So, for that configuration the functional coverage tool must not try to check for those states or transitions. This is accomplished by making the tool configurable through TPP.

Similarly, there are monitors that check for illegal conditions occurring within the HDL simulation. These illegal conditions could show up as bugs. For example on a three-state bus, 2 drivers should not be on simultaneously. These monitors are configurable – adding or removing checks based on whether a particular logic is included or not for that configuration.

The cosimulation mechanism connects the HDL to the ISS 126. It is used to check that the state of the processor at the end of the instruction is identical in the HDL and the ISS 126. It too is configurable to the extent that it knows what features are included for each configuration and what state needs to be compared. So, for example, the data breakpoint feature adds a special register. This mechanism needs to know to compare this new special register.

Instruction semantics specified via TIE can be translated to functionally equivalent C functions for use in the ISS 126 and for system designers to use for testing and verification. The semantics of an instruction in the configuration database of search engine 106 are translated to a C function by tools that build a parse tree using standard parser tools, and then code that walks the tree and outputs the corresponding expressions in the C language. The translation requires a prepass to assign bit widths to all expressions and to rewrite the parse tree to simplify some translations. These translators are relatively simple compared to other translators, such as HDL to C or C to assembly language compilers, and can be written by one skilled in the art starting from the TIE and C language specification.

Using a compiler configured using the configuration file 100 and the assembler/disassembler 100, benchmark application source code 118 is compiled and assembled and, using a sample data set 124, simulated to obtain a software profile 130 which also is provided to the user configuration capture routine for feedback to the user.

Having the ability to obtain both the hardware and software cost/benefit characterizations for any configuration parameter selections opens up new opportunities for further optimization of the system by the designers. Specifically, this will enable designers to select the optimal configuration parameters which optimize the overall systems according to some figure of merit. One possible process is based on a greedy strategy, by repeatedly selecting or de-selecting a configuration parameter. At each step, the parameter that has the best impact on the overall system performance and cost is selected. This step is repeated until no single parameter can be changed to improve the system performance and cost. Other



extensions include looking at a group of configuration parameters at a time or employing more sophisticated searching algorithms.

In addition to obtaining optimal configuration parameter selection, this process can also be used to construct optimal processor extensions. Because of the large number of possibilities in the processor extensions, it is important to restrict the number of extension candidates. One technique is to analyze the application software and only look at the instruction extensions that can improve the system performance or cost.

Having covered the operation of an automated processor configuration system according to this embodiment, examples now will be given of application of the system to processor microarchitecture configuration. The first example shows the advantages of the present invention as applied to image compression.

R Motion estimation is an important component of many image compression algorithms, including MPEG video and H263 conference applications. Video image compression attempts to use the similarities from one frame to the next to reduce the amount of storage required for each frame. In the simplest case, each block of an image to be compressed can be compared to the corresponding block (the same X,Y location) of the reference image (one that closely precedes or follows the image being compressed). The compression of the image differences between frames is generally more bit-efficient than compression of the individual images. In video sequences, the distinctive image features often move from frame to frame, so the closest correspondence between blocks in different frames is often not at exactly the same X,Y location, but at some offset. If significant parts of the image are moving between frames, it may be necessary to identify and compensate for the movement, before computing the difference. This fact means that the densest representation can be achieved by encoding the difference between successive images, including, for distinctive features, an X, Y offset in the sub-images used in the computed difference. The offset in the location used for computing the image difference is called the motion vector.

The most computationally intensive task in this kind of image compression is the determination of the most appropriate motion vector for each block. The common metric for selecting the motion vector is to find the vector with the lowest average pixel-by-pixel difference between each block of the image being compressed and a set of candidate blocks of the previous image. The candidate blocks are the set of all the blocks in a neighborhood around the location of the block being compressed. The size of the image, the size of the block and size of the neighborhood all affect the running time of the motion estimation algorithm.

Simple block-based motion estimation compares each sub-image of the image to be compressed against a reference image. The reference image may precede or follow the subject image in the video sequence. In every case, the reference image is known to be available to the decompression system before the subject image is decompressed. The comparison of one block of an image under compression with candidate blocks of a reference image is illustrated below.

For each block in the subject image, a search is performed around the corresponding location in the reference image. Normally each color component (e.g., YUV) of the images is analyzed separately. Sometimes motion estimation is performed only on one component, especially luminance. The average pixel-by-pixel difference is computed between that subject block and every possible block in the search zone of the reference image. The difference is the absolute value of the difference in magnitude of the pixel values. The average is proportional to the sum over the $N^2$ pixels in the pair of blocks (where N is the dimension of the block). The block of the reference image that produces the smallest average pixel difference defines the motion vector for that block of the subject image.

The following example shows a simple form of a motion estimation algorithm, then optimizes the algorithm using TIE for a small application-specific functional unit. This optimization yields a speed-up of more than a factor of 10, making processor-based compression feasible for many video applications. It illustrates the power of a configurable processor that combines the ease of programming in a high-level language with the efficiency of special-purpose hardware.

This example uses two matrices, OldB and NewB, to respectively represent the old and new images. The size of the image is determined by NX and NY. The block size is determined by BLOCKX and BLOCKY. Therefore, the image is composed of NX/ BLOCKX by NY/BLOCKY blocks. The search region around a block is determined by SEARCHX and SEARCHY. The best motion vectors and values are stored in VectX, VectY, and VectB. The best motion vectors and values computed by the base (reference) implementation are stored in BaseX, BaseY, and BaseB. These values are used to check against the vectors computed by the implementation using instruction extensions. These basic definitions are captured in the following C-code segment:

```
#define NX 64                                     /* image width */
#define NY 32                                     /* image height */
#define BLOCKX 16                                 /* block width */
#define BLOCKY 16                                 /* block height */
#define SEARCHX 4                                 /* search region
                                                  width */
#define SEARCHY 4                                 /* search region
                                                  height */
unsigned char OldB[NX][NY];                       /* old image */
unsigned char NewB[NX][NY];                       /* new image */
unsigned short VectX[NX/BLOCKX][NY/BLOCKY]; /* X motion vector */
unsigned short VectY[NX/BLOCKX][NY/BLOCKY]; /* Y motion vector */
unsigned short VectB[NX/BLOCKX][NY/BLOCKY]; /* absolute
                                                  difference */
unsigned short BaseX[NX/BLOCKX][NY/BLOCKY]; /* Base X motion
                                                  vector */
unsigned short BaseY[NX/BLOCKX][NY/BLOCKY]; /* Base Y motion
                                                  vector */
unsigned short BaseB[NX/BLOCKX][NY/BLOCKY]; /* Base absolute
                                                  difference*/
#define ABS(x)      (((x) < 0) ? (-(x)) : (x))
#define MIN(x,y)    (((x) < (y)) ? (x) : (y))
#define MAX(x,y)    (((x) > (y)) ? (x) : (y))
```



```
#define ABSD(x,y)  (((x) > (y)) ? ((x) - (y)) : ((y) - (x)))
```

The motion estimation algorithm is comprised of three nested loops:

1. For each source block in the old image.
2. For each destination block of the new image in the surrounding region of the source block.
3. Compute the absolute difference between each pair of pixels.

The complete code for the algorithm is listed below.

```
/************************************************************
    Reference software implementation
************************************************************/
void
motion_estimate_base()
{
    int bx, by, cx, cy, x, y;
    int startx, starty, endx, endy;
    unsigned diff, best, bestx, besty;
    for(bx = 0; bx < NX/BLOCKX; bx++) {
        for(by = 0; by < NY/BLOCKY; by++) {
            best = bestx = besty = UINT_MAX;
            startx = MAX(0, bx*BLOCKX - SEARCHX);
            starty = MAX(0, by*BLOCKY - SEARCHY);
            endx = MIN(NX-BLOCKX, bx*BLOCKX + SEARCHX);
            endy = MIN(NY-BLOCKY, by*BLOCKY + SEARCHY);
            for(cx = startx; cx < endx; cx++) {
                for(cy = starty; cy < endy; cy++) {
                    diff = 0;
                    for(x = 0; x < BLOCKX; x++) {
                        for(y = 0; y < BLOCKY; y++) {
                            diff += ABSD(OldB[cx+x][cy+y],
                                 NewB[bx*BLOCKX+x][by*BLOCKY+y]);
                        }
                    }
                    if (diff < best) {
                        best = diff;
                        bestx = cx;
                        besty = cy;
                    }
                }
            }
            BaseX[bx][by] = bestx;
            BaseY[bx][by] = besty;
            BaseB[bx][by] = best;
        }
    }
}
```

While the basic implementation is simple, it fails to exploit much of the intrinsic parallelism of this block to block comparison. The configurable processor architecture provides two key tools to allow significant speed-up of this application.

First, the instruction set architecture includes powerful funnel shifting primitives to permit rapid extraction of unaligned fields in memory. This allows the inner loop of the pixel comparison to fetch groups of adjacent pixels from memory efficiently. The loop can then be rewritten to operate on four pixels (bytes) simultaneously. In particular, for the purposes of this example it is desirable to define a new instruction to compute the absolute difference of four pixel pairs at a time. Before defining this new instruction, however, it is necessary to re-implement the algorithm to make use of such an instruction.

The presence of this instruction allows such improvement in the inner loop pixel difference computation that loop unrolling becomes attractive as well. The C code for the inner loop is rewritten to take advantage of the new sum-of-absolute-differences instruction and the efficient shifting. Part of four overlapping blocks of the reference image can then be compared in the same loop. `SAD(x,y)` is the new intrinsic function corresponding to the added instruction. `SRC(x, y)` performs a right shift of the concatenation of x and y by the shift amount stored in the SAR register.

```
/*****************************************************************
    Fast version of motion estimation which uses the SAD
        instruction.
*****************************************************************/
void
motion_estimate_tie()
{
    int bx, by, cx, cy, x;
    int startx, starty, endx, endy;
    unsigned diff0, diff1, diff2, diff3, best, bestx, besty;
    unsigned *N, N1, N2, N3, N4, *O, A,B,C,D,E;
    for(bx = 0; bx < NX/BLOCKX; bx++) {
        for(by = 0; by < NY/BLOCKY; by++) {
            best = bestx = besty = UINT_MAX;
            startx = MAX(0, bx*BLOCKX - SEARCHX);
            starty = MAX(0, by*BLOCKY - SEARCHY);
            endx = MIN(NX-BLOCKX, bx*BLOCKX + SEARCHX);
            endy = MIN(NY-BLOCKY, by*BLOCKY + SEARCHY);
            for(cy = starty; cy < endy; cy += sizeof(long)) {
                for(cx = startx; cx < endx; cx++) {
                    diff0 = diff1 = diff2 = diff3 = 0;
                    for(x = 0; x < BLOCKX; x++) {
                        N = (unsigned *) & (NewB[bx*BLOCKX+x]
                            [by*BLOCKY]);
                        N1 = N[0];
                        N2 = N[1];
                        N3 = N[2];
                        N4 = N[3];
                        O = (unsigned *) &(OldB[cx+x][cy]);
                        A = O[0];
                        B = O[1];
                        C = O[2];
                        D = O[3];
                        E = O[4];
                        diff0 += SAD(A, N1) + SAD(B, N2) +
                                  SAD(C, N3) + SAD(D, N4);
                        SSAI(8);
```



```
                        diff1 += SAD(SRC(B,A), N1) +
                            SAD(SRC(C,B), N2) + SAD(SRC(D,C),
                            N3) + SAD(SRC(E,D), N4);
                      SSAI(16);
                        diff2 += SAD(SRC(B,A), N1) +
                            SAD(SRC(C,B), N2) + SAD(SRC(D,C),
                            N3) + SAD(SRC(E,D), N4);
                        SSAI(24);
                        diff3 += SAD(SRC(B,A), N1) +
                            SAD(SRC(C,B), N2) + SAD(SRC(D,C),
                            N3) + SAD(SRC(E,D), N4);
                        O += NY/4;
                        N += NY/4;
                    }
                    if (diff0 < best) {
                        best = diff0;
                        bestx = cx;
                        besty = cy;
                    }
                    if (diff1 < best) {
                        best = diff1;
                        bestx = cx;
                        besty = cy + 1;
                    }
                    if (diff2 < best) {
                        best = diff2;
                        bestx = cx;
                        besty = cy + 2;
                    }
                    if (diff3 < best) {
                        best = diff3;
                        bestx = cx;
                        besty = cy + 3;
                    }

                }
            }
            VectX[bx][by] = bestx;
            VectY[bx][by] = besty;
            VectB[bx][by] = best;
        }
    }
}
```

This implementation uses the following SAD function to emulate the eventual new instruction:

```
/*****************************************************************
    Sum of absolute difference of four bytes
*****************************************************************/
static inline unsigned
SAD(unsigned ars, unsigned art)
{
    return ABSD(ars >> 24, art >> 24) +
          ABSD((ars >> 16) & 255, (art >> 16) & 255) +
```

```
            ABSD((ars >> 8) & 255, (art >> 8) & 255) +
            ABSD(ars & 255, art & 255);
    }
```

To debug this new implementation, the following test program is used to compare the motion vectors and values computed by the new implementation and the base implementation:

```
/*****************************************************************
    Main test
*****************************************************************/
int
main(int argc, char **argv)
{
    int passwd;
#ifndef NOPRINTF
    printf("Block=(%d,%d), Search=(%d,%d), size=(%d,%d)\n",
        BLOCKX, BLOCKY, SEARCHX, SEARCHY, NX, NY);
#endif
    init();
    motion_estimate_base();
    motion_estimate_tie();
    passwd = check();
#ifndef NOPRINTF
    printf(passwd ? "TIE version passed\n" : "*** TIE version
failed\n");
#endif

    return passwd;
}
```

This simple test program will be used throughout the development process. One important convention that must be followed here is that the main program must return 0 when an error is detected and 1 otherwise.

The use of TIE permits rapid specification of new instructions. The configfurable processor generator can fully implement these instructions in both the hardware implementation and the software development tools. Hardware synthesis creates an optimal integration of the new function into the hardware datapath. The configurable processor software environment fully supports the new instructions in the C and C++ compilers, the assembler, the symbolic debugger, the profiler and the cycle-accurate instruction set simulator. The rapid regeneration of hardware and software makes application-specific instructions a quick and reliable tool for application acceleration.

This example uses TIE to implement a simple instruction to perform pixel differencing, absolute value and accumulation on four pixels in parallel. This single instruction does eleven basic operations (which in a conventional process might require separate instructions) as an atomic operation. The following is the complete description:

```
// define a new opcode for Sum of Absolute Difference (SAD)
// from which instruction decoding logic is derived
```



```
opcode SAD op2=4'b0000 CUST0
// define a new instruction class
// from which compiler, assembler, disassembler
// routines are derived
iclass sad {SAD} {out arr, in ars, in art}
// semantic definition from which instruction-set
// simulation and RTL descriptions are derived
semantic sad_logic {SAD} {
    wire [8:0] diff01, diff11, diff21, diff31;
    wire [7:0] diff0r, diff1r, diff2r, diff3r;
    assign diff01 = art[7:0] - ars[7:0];
    assign diff11 = art[15:8] - ars[15:8];
    assign diff21 = art[23:16] - ars[23:16];
    assign diff31 = art[31:24] - ars[31:24];
    assign diff0r = ars[7:0] - art[7:0];
    assign diff1r = ars[15:8] - art[15:8];
    assign diff2r = ars[23:16] - art[23:16];
    assign diff3r = ars[31:24] - art[31:24];
    assign arr =
        (diff01[8] ? diff0r : diff01) +
        (diff11[8] ? diff1r : diff11) +
        (diff21[8] ? diff2r : diff21) +
        (diff31[8] ? diff3r : diff31);
}
```

This description represents the minimum steps needed to define a new instruction. First, it is necessary to define a new opcode for the instruction. In this case, the new opcode SAD is defined as a sub-opcode of CUST0. As noted above, CUST0 is predefined as:

```
opcode QRST   op0=4'b0000
opcode CUST0  op1=4'b0100   QRST
```

It is easy to see that QRST is the top-level opcode. CUST0 is a sub-opcode of QRST and SAD in turn is a sub-opcode of CUST0. This hierarchical organization of opcodes allow logical grouping and management of the opcode spaces. One important thing to remember is that CUST0 (and CUST1) are defined as reserved opcode space for users to add new instructions. It is preferred that users stay within this allocated opcode space to ensure future re-usability of TIE descriptions.

The second step in this TIE description is to define a new instruction class containing the new instruction SAD. This is where the operands of SAD instruction is defined. In this case, SAD consists of three register operands, destination register `arr` and source registers `ars` and `art`. As noted previously, `arr` is defined as the register indexed by the r field of the instruction, `ars` and `art` are defined as registers indexed by the s and t fields of the instruction.

The last block in this description gives the formal semantic definitions for the SAD instruction. The description is using a subset of Verilog HDL language for describing combination logic. It is this block that defines precisely how the ISS will simulate the SAD instruction and how an additional circuitry is synthesized and added to the configurable processor hardware to support the new instruction.

Next, the TIE description is debugged and verified using the tools previously described. After verifying the correctness of the TIE description, the next step is to estimate the impact of the new instruction on the hardware size and performance. As noted above, this can be done using, e.g., Design Compiler™. When Design Compiler finishes, the user can look at the output for detailed area and speed reports.

After verifying that the TIE description is correct and efficient, it is time to configure and build a configurable processor that also supports the new SAD instruction. This is done using the GUI as described above.

Next, the motion estimation code is compiled into code for the configurable processor which uses the instruction set simulator to verify the correctness of the program and more importantly to measure the performance. This is done in three steps: run the test program using the simulator; run just the base implementation to get the instruction count; and run just the new implementation to get the instruction count.

The following is the simulation output of the second step:

```
Block=(16,16), Search=(4,4), size=(32,32)
TIE version passed
Simulation Completed Successfully
Time for Simulation = 0.98 seconds
Events                           Number  Number
                                        per 100
                                         instrs
Instructions                     226005 ( 100.00 )
Unconditional taken branches        454 (   0.20 )
Conditional branches              37149 (  16.44 )
     Taken                        26947 (  11.92 )
     Not taken                    10202 (   4.51 )
Window Overflows                     20 (   0.01 )
Window Underflows                    19 (   0.01 )
```

The following is the simulation output of the last step:

```
Block=(16,16), Search=(4,4), size=(32,32)
TIE version passed
Simulation Completed Successfully
Time for Simulation = 0.36 seconds
Events                           Number  Number
                                        per 100
                                         instrs
Instructions                      51743 ( 100.00 )
Unconditional taken branches        706 (   1.36 )
Conditional branches               3541 (   6.84 )
     Taken                         2759 (   5.33 )
     Not taken                      782 (   1.51 )
Window Overflows                     20 (   0.04 )
Window Underflows                    19 (   0.04 )
```



From the two reports one can see that roughly a 4x speedup has occurred. Notice that the the configurable processor instruction set simulator can provide much other useful information.

After verifying the correctness and performance of the program, the next step is to run the test program using a Verilog simulator as described above. Those skilled in the art can glean the details of this process from the makefile of Appendix C (associated files also are shown in Appendix C). The purpose of this simulation is to further verify the correctness of the new implementation and more importantly to make this test program as part of the regression test for this configured processor.

Finally, the processor logic can be synthesized using, e.g., Design Compiler™ and placed and routed using, e.g., Apollo™.

This example has taken a simplified view of video compression and motion estimation for the sake of clarity and simplicity of explanation. In reality, there are many additional nuances in the standard compression algorithms. For example, MPEG 2 typically does motion estimation and compensation with sub-pixel resolution. Two adjacent rows or columns of pixels can be averaged to create a set of pixels interpolated to an imaginary position halfway between the two rows or columns. The configurable processor's user-defined instructions are again useful here, since a parallel pixel averaging instruction is easily implemented in just three or four lines of TIE code. Averaging between pixels in a row again uses the efficient alignment operations of the processor's standard instruction set.

Thus, the incorporation of a simple sum-of-absolute-differences instruction adds just a few hundred gates, yet improves motion estimation performance by more than a factor of ten. This acceleration represents significant improvements in cost and power efficiency of the final system. Moreover, the seamless extension of the software development tools to include the new motion-estimation instruction allows for rapid prototyping, performance analysis and release of the complete software application solution. The solution of the present invention makes application-specific processor configuration simple, reliable and complete, and offers dramatic enhancement of the cost, performance, functionality and power-efficiency of the final system product.

As an example focusing on the addition of a functional hardware unit, consider the base configuration shown in FIG. 6 which includes the processor control function, program counter (PC), branch selection, instruction memory or cache and instruction decoder, and the basic integer datapath including the main register file, bypassing multiplexers, pipeline registers, ALU, address generator and data memory for the cache.

The HDL is written with the presence of the multiplier logic being conditional upon the "multiplier" parameter being set, and a multiplier unit is added as a new pipeline stage as shown in FIG. 7 (changes to exception handling may be required if precise exceptions are to be supported). Of course, instructions for making use of the multiplier are preferably added concomitantly with the new unit.

As a second example, a full coprocessor may be added to the base configuration as shown in FIG. 8 for a digital signal processor such as a multiply/accumulate unit. This entails changes in processor control such as adding decoding control signals for multiply-accumulate operations, including decoding of

register sources and destinations from extended instructions; adding appropriate pipeline delays for control signals; extending register destination logic; adding control for a register bypass multiplexer for moves from accumulate registers, and the inclusion of a multiply-accumulate unit as a possible source for an instruction result. Additionally, it requires addition of a multiply-accumulate unit which entails additional accumulator registers, a multiply-accumulate array and source select multiplexers for main register sources. Also, addition of the coprocessor entails extension of the register bypass multiplexer from the accumulate registers to take a source from the accumulate registers, and extension of the load/alignment multiplexer to take a source from the multiplier result. Again, the system preferably adds instructions for using the new functional unit along with the actual hardware.

Another option that is particularly useful in connection with digital signal processors is a floating point unit. Such a functional unit implementing, e.g., the IEEE 754 single-precision floating point operation standard may be added along with instructions for accessing it. The floating point unit may be used, e.g., in digital signal processing applications such as audio compression and decompression.

As yet another example of the system's flexibility, consider the 4 kB memory interface shown in FIG. 9. Using the configurability of the present invention, coprocessor registers and datapaths may be wider or narrower than the main integer register files and datapaths, and the local memory width may be varied so that the memory width is equal to the widest processor or coprocessor width (addressing of memory on reads and writes being adjusted accordingly). For example, FIG. 10 shows a local memory system for a processor that supports loads and stores of 32 bits to a processor/coprocessor combination addressing the same array, but where the coprocessor supports loads and stores of 128 bits. This can be implemented using the TPP code

```
function memory(Select,A1,A2,DI1,DI2,W1,W2,DO1,DO2)
;    $B1    =    config_get_value("width_of_port_1");    $B2    =
config_get_value("width_of_port_2");
; $Bytes = config_get_value("size_of_memory");
; $Max = max($B1,$B2); $Min = min($B1,$B2);
; $Banks = $Max/$Min;
; $Wide1 = ($Max == $B1); $Wide2 = ($Max == $B2);
; $Depth = $Bytes/(log2($Banks)*log2($Max));

wire [`$Max`*8-1:0] Data1 = `$Wide1`?DI1:{`$Banks`{DI1}};
wire [`$Max`*8-1:0] Data2 = `$Wide1`?DI2:{`$Banks`{DI2}});
wire [`$Max`*8-1:0] D = Select ? Data1 : Data2;
wire Wide = Select ? Wide1: Wide2;
wire [log2(`$Bytes`)-1:0] A = Select? A1 : A2;
wire    [log2(`$Bytes`)-1:0]    Address    =    A[log2(`$Bytes`)-
1:log2(`$Banks`)]:
wire [log2(`$Banks`)-1:0] Lane = A[log2(`$Banks`)-1:0];
;for ($i=0; $i<$Banks; $i++) {
  wire WrEnable(i) = Wide | (Lane == (i));
  wire    [log2(`$Min`)-1:0]    WrData`$i`    =    D[((i)+1)*`$Min`*8-
1:(i)*`$Min`*8]
  ram(RdData`$i`,Depth,Address,WrData`$i`,WrEnable`$i`);
;}
wire [`$Max`*8-1:0] RdData = {
```



```
;for ($i=0; $i<$Banks; $i++) {
  RdData`$i`,
;}
  }
wire   [`$B1`*8-1:0]   DO1   =   Wide1?RdData:RdData[(Lane+1)*B1*8-
1:Lane*B1*8];
wire   [`$B2`*8-1:0]   DO2   =   Wide2?RdData:RdData[(Lane+1)*B2*8-
1:Lane*B2*8];
```

where `$Bytes` is the total memory size accessed either as width B1 bytes at byte address A1 with data bus D1 under control of write signal W1, or using corresponding parameters B2, A2, D2 and W2. Only one set of signals, defined by `Select`, is active in a given cycle. The TPP code implements the memory as a collection of memory banks. The width of each bank is given by the minimum access width and the number of banks by the ratio of the maximum and minimum access widths. A for loop is used to instantiate each memory bank and its associated write signals, i.e., write enable and write data. A second for loop is used to gather the data read from all the banks into a single bus.

FIG. 11 shows an example of the inclusion of user-defined instructions in the base configuration. As shown in the Figure, simple instructions may be added to the processor pipeline with timing and interface similar to that of the ALU. Instructions added in this way must generate no stalls or exceptions, contain no state, use only the two normal source register values and the instruction word as inputs, and generate a single output value. If, however, the TIE language has provisions for specifying processor state, such constraints are not necessary.

FIG. 12 shows another example of implementation of a user-defined unit under this system. The functional unit shown in the Figure, an 8/16 parallel data unit extension of the ALU, is generated from the following ISA code:

```
Instruction {
Opcode     ADD8_4     CUSTOM     op2=0000
Opcode     MIN16_2    CUSTOM     op2=0001
Opcode     SHIFT16_2  CUSTOM     op2=0002
iclass          MY    4ADD8,2MIN16,SHIFT16_2          a<t,a<s,a>t
}
Implementation {
input [31:0] art, ars;
input [23:0] inst;
input ADD8_4, MIN16_2, SHIFT16_2;
output [31:0] arr;
wire [31:0] add, min, shift;
assign add = {art[31:24] + ars[31:24], art[23:16] + art[23:16],
art[15:8] + art[15:8], art[7:0] + art[7:0]};
assign  min[31:16]  =  art[31:16]  <  ars[31:16]  ?  Art[31:16]  :
ars[31:16];
assign min[15:0] = art[15:0] < ars[15:0] ? Art[15:0] : ars[15:0];
assign shift[31:16] = art[31:16] << ars[31:16];
assign shift[15:0] = art[15:0] << ars[15:0];
assign  arr  =  {32{ADD8_4}}  &  add  |  {32{MIN16_2}}  &  min  |
{32{SHIFT16_2}} & shift;
```

}

Of particular interest in another aspect of the present invention is the designer-defined instruction execution unit 96, for it is there that TIE-defined instructions, including those modifying processor state, are decoded and executed. In this aspect of the invention, a number of building blocks have been added to the language to make it possible to declare additional processor states which can be read and written by the new instructions. These "state" statements are used to declare the addition processor states. The declaration begins with the keyword state. The next section of the state statements describes the size, number of bits, of the state and how the bits of the states are indexed. The section following that is the name of the state, used to identify the state in other description sections. The last section of the "state" statement is a list of attributes associated with the state. For example,

```
state [63:0] DATA cpn=0 autopack
state [27:0] KEYC cpn=1 nopack
state [27:0] KEYD cpn=1
```

defines three new processor states, DATA, KEYC, and KEYD. State DATA is 64-bits wide and the bits are indexed from 63 to 0. KEYC and KEYD are both 28-bit states. DATA has a coprocessor-number attribute cpn indicating to which coprocessor data DATA belongs.

The attribute "autopack" indicate that the state DATA will be automatically mapped to some registers in the user-register file so that the value of DATA can be read and written by software tools.

The `user_register` section is defined to indicate the mapping of states to registers in the user register file. A `user_register` section starts with a keyword `user_register`, followed by a number indicating the register number, and ends with an expression indicating the state bits to be mapped onto the register. For example,

```
user_register 0 DATA[31:0]
user_register 1 DATA[63:32]
user_register 2 KEYC
user_register 3 KEYD
user_register 4 {X, Y, Z}
```

specifies that the low-order word of DATA is mapped to the first user register file and the high order word to the second. The next two user register file entries are used to hold values of KEYC and KEYD. Clearly, the state information used in this section must be consistent with that of the `state` section. Here, the consistency can be checked automatically by a computer program.

In another embodiment of the present invention, such an assignment of state bits to user register file entries is derived automatically using bin-packing algorithms. In yet another embodiment, a combination of manual and automatic assignments can be used, for example, to ensure upward compatibility.

Instruction field statements `field` are used to improve the readability of the TIE code. Fields are subsets or concatenations of other fields that are grouped together and referenced by a name. The



complete set of bits in an instruction is the highest-level superset field `inst`, and this field can be divided into smaller fields. For example,

```
field x    inst[11:8]
field y    inst[15:12]
field xy   {x, y}
```

defines two 4-bit fields, x and y, as sub-fields (bits 8-11 and 12-15, respectively) of a highest-level field `inst` and an 8-bit field xy as the concatenation of the x and y fields.

The statements `opcode` define opcodes for encoding specific fields. Instruction fields that are intended to specify operands, e.g., registers or immediate constants, to be used by the thus-defined opcodes, must first be defined with field statements and then defined with operand statements.

For example,

```
opcode     acs   op2 = 4'b0000     CUST0
opcode     adsel op2 = 4'b0001     CUST0
```

defines two new opcodes, `acs` and `adsel`, based on the previously-defined opcode CUST0 (4'b0000 denotes a four bit-long binary constant 0000). The TIE specification of the preferred core ISA has the statements

```
field op0   inst[3:0]
field op1   inst[19:16]
field op2   inst[23:20]
opcode      QRST  op0 = 4'b0000
opcode      CUST0 op1=4'b0100 QRST
```

as part of its base definitions. Thus, the definitions of `acs` and `adsel` cause the TIE compiler to generate instruction decoding logic respectively represented by the following:

```
inst[23:0] = 0000 0110 xxxx xxxx xxxx 0000
inst[23:0] = 0001 0110 xxxx xxxx xxxx 0000
```

Instruction operand statements `operand` identify registers and immediate constants. Before defining a field as an operand, however, it must have been previously been defined as a field as above. If the operand is an immediate constant, the value of the constant can be generated from the operand, or it can be taken from a previously defined constant table defined as described below. For example, to encode an immediate operand the TIE code

```
field          offset    inst[23:6]
operand        offests4  offset {
     assign    offsets4 = {{14{offset[17]}}, offset}<<2;
}{
     wire      [31:0] t;
   assign  t = offsets4>>2;
   assign  offset = t[17:0];
}
```

defines an 18-bit field named offset which holds a signed number and an operand `offsets4` which is four times the number stored in the offset field. The last part of the `operand` statement actually

describes the circuitry used to perform the computations in a subset of the Verilog™ HDL for describing combinatorial circuits, as will be apparent to those skilled in the art.

Here, the `wire` statement defines a set of logical wires named `t` thirty-two bits wide. The first `assign` statement after the wire statement specifies that the logical signals driving the logical wires are the `offsets4` constant shifted to the right, and the second `assign` statement specifies that the lower eighteen bits of `t` are put into the `offset` field. The very first `assign` statement directly specifies the value of the `offsets4` operand as a concatenation of `offset` and fourteen replications of its sign bit (bit 17) followed by a shift-left of two bits.

For a constant table operand, the TIE code

```
table prime 16 {
    2, 3, 5, 7, 9, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
53
}
operand         prime_s     s {
    assign      prime_s = prime[s];
} {
    assign s = prime_s == prime[0]   ?    4'b0000 :
               prime_s == prime[1]   ?    4'b0001 :
               prime_s == prime[2]   ?    4'b0010 :
               prime_s == prime[3]   ?    4'b0011 :
               prime_s == prime[4]   ?    4'b0100 :
               prime_s == prime[5]   ?    4'b0101 :
               prime_s == prime[6]   ?    4'b0110 :
               prime_s == prime[7]   ?    4'b0111 :
               prime_s == prime[8]   ?    4'b1000 :
               prime_s == prime[9]   ?    4'b1001 :
               prime_s == prime[10]  ?    4'b1010 :
               prime_s == prime[11]  ?    4'b1011 :
               prime_s == prime[12]  ?    4'b1100 :
               prime_s == prime[13]  ?    4'b1101 :
               prime_s == prime[14]  ?    4'b1110 :
                                          4'b1111;
}
```

makes use of the `table` statement to define an array `prime` of constants (the number following the table name being the number of elements in the table) and uses the operand `s` as an index into the table `prime` to encode a value for the operand `prime_s` (note the use of Verilog™ statements in defining the indexing).

The instruction class statement `iclass` associates opcodes with operands in a common format. All instructions defined in an `iclass` statement have the same format and operand usage. Before defining an instruction class, its components must be defined, first as fields and then as opcodes and operands. For example, building on the code used in the preceding example defining opcodes `acs` and `adsel`, the additional statements

```
operand     art     t     {assign art = AR[t];} {}
operand     ars     s     {assign ars = AR[s];} {}
```



```
operand     arr     r     {assign AR[r] = arr;} {}
```

use the `operand` statement to define three register operands `art`, `ars` and `arr` (again note the use of Verilog™ statements in the definition). Then, the iclass statement

```
iclass      viterbi     {adsel, acs} {out arr, in art, in ars}
```

specifies that the operands `adsel` and `acs` belong to a common class of instructions `viterbi` which take two register operands `art` and `ars` as input and writes output to a register operand `arr`.

In the present invention, the instruction class statement "`iclass`" is modified to allow the specification of state-access information of instructions. It starts with a keyword "`iclass`", is followed by the name of the instruction class, the list of opcodes belonging to the instruction class and a list of operand access information, and ends with a newly-defined list for state access information. For example,

```
iclass lddata {LDDATA} {out arr, in imm4} {in DATA}
iclass stdata {STDATA} {in ars, in art}   {out DATA}
iclass stkey  {STKEY}  {in ars, in art}   {out KEYC, out KEYD}
iclass des    {DES}    {out arr, in imm4} {inout KEYC, inout DATA,
inout KEYD}
```

defines several instruction classes and how various new instructions access the states. The keywords "`in`", "`out`", and "`inout`" are used to indicate that the state is read, written, or modified (read and written) by the instructions in the `iclass`. In this example, state "DATA" is read by the instruction "LDDATA", state "KEYC" and "KEYD" are written by the instruction "STKEY", and "KEYC", "KEYD", and "DATA" are modified by the instruction "DES".

The instruction semantic statement `semantic` describes the behavior of one or more instructions using the same subset of Verilog™ used for coding operands. By defining multiple instructions in a single semantic statement, some common expressions can be shared and the hardware implementation can be made more efficient. The variables allowed in semantic statements are operands for opcodes defined in the statement's opcode list, and a single-bit variable for each opcode specified in the opcode list. This variable has the same name as the opcode and evaluates to 1 when the opcode is detected. It is used in the computation section (the Verilog™ subset section) to indicate the presence of the corresponding instruction.

```
// define a new opcode for BYTESWAP based on
//    - a predefined instruction field op2
//    - a predefined opcode CUST0
// refer to Xtensa ISA manual for descriptions of op2 and CUST0
opcode   BYTESWAP   op2=4'b0000   CUST0

// declare state SWAP and COUNT
state COUNT     32
state SWAP      1

// map COUNT and SWAP to user register file entries
```

```
user_register 0 COUNT
user_register 1 SWAP

// define a new instruction class that
//    - reads data from ars (predefined to be AR[s])
//    - uses and writes state COUNT
//    - uses state SWAP
iclass  bs  {BYTESWAP}   {out arr, in ars} {inout COUNT, in SWAP}

// semantic definition of byteswap
//    COUNT the number of byte-swapped words
//    Return the swapped or un-swapped data depending on SWAP
semantic bs {BYTESWAP} {
    wire             [31:0]              ars_swapped             =
{ars[7:0],ars[15:8],ars[23:16],ars[31:24]};
    assign arr = SWAP ? ars_swapped : ars;
    assign COUNT = COUNT + SWAP;
}
```

The first section of the above code defines an opcode for the new instruction, called BYTESWAP.

```
// define a new opcode for BYTESWAP based on
//    - a predefined instruction field op2
//    - a predefined opcode CUSTO
// refer to Xtensa ISA manual for descriptions of op2 and CUSTO
opcode  BYTESWAP  op2=4'b0000  CUSTO
```

Here, the new opcode BYTESWAP is defined as a sub-opcode of CUSTO. From the Xtensa™ Instruction Set Architecture Reference Manual described in greater detail below, one sees that CUSTO is defined as

```
opcode QRST op0=4'b0000
opcode CUSTO op1=4'b0100 QRST
```

where op0 and op1 are fields in the instruction. Opcodes are typically organized in a hierarchical fashion. Here, QRST is the top-level opcode and CUSTO is a sub-opcode of QRST and BYTESWAP is in turn a sub-opcode of CUSTO. This hierarchical organization of opcodes allows logical grouping and management of the opcode spaces.

The second declaration declares additional processor states needed by the BYTESWAP instruction:

```
// declare state SWAP and COUNT
state COUNT     32
state SWAP    1
```

Here, COUNT is declared as a 32-bit state and SWAP as a 1-bit state. The TIE language specifies that the bits in COUNT are indexed from 31 to 0 with bit 0 being least significant.

The Xtensa™ ISA provides two instructions, RSR and WSR, for saving and restoring special system registers. Similarly, it provides two other instructions, RUR and WUR (described in greater detail below) for saving and restoring states which are declared in TIE. In order to save and restore states



declared in TIE, one must specify the mapping of the states to entries in the user register file that RUR and . WUR instructions can access. The following section of the above code specifies this mapping:

```
// map COUNT and SWAP to user register file entries
user_register 0 COUNT
user_register 1 SWAP
```

such that the following instructions will save the value of COUNT to a2 and SWAP to a5:

```
RUR a2, 0;
RUR a5, 1;
```

This mechanism is actually used in the test program to verify the contents of the states. In C, the above two instructions would look like:

```
x = RUR(0);
y = RUR(1);
```

The nest section in the TIE description is the definition of a new instruction class containing the new instruction BYTESWAP:

```
// define a new instruction class that
//    - reads data from ars (predefined to be AR[s])
//    - uses and writes state COUNT
//    - uses state SWAP
iclass  bs  {BYTESWAP}   {out arr, in ars} {inout COUNT, in SWAP}
```

where iclass is the keyword and bs is the name of the iclass. The next clause lists the instruction in this instruction class (BYTESWAP). The clause after than specifies the operands used by the instructions in this class (in this case an input operand ars and an output operand arr). The last clause in the iclass definition specifies the states which are accessed by the instruction in this class (in this case the instruction will read state SWAP and read and write state COUNT).

The last block in the above code gives the formal semantic definition for the BYTESWAP instruction:

```
// semantic definition of byteswap
//    COUNT the number of byte-swapped words
//    Return the swapped or un-swapped data depending on SWAP
semantic bs {BYTESWAP} {
    wire             [31:0]              ars_swapped             =
{ars[7:0],ars[15:8],ars[23:16],ars[31:24]};
    assign arr = SWAP ? ars_swapped : ars;
    assign COUNT = COUNT + SWAP;
}
```

The description uses a subset for Verilog HDL for describing combination logic. It is this block that defines precisely how the instruction set simulator will simulate the BYTESWAP instruction and how the

additional circuitry is synthesized and added to the Xtensa™ processor hardware to support the new instruction.

In the present invention implementing user-defined states, the declared states can be used just like any other variables for accessing information stored in the states. A state identifier appearing on the right hand side of an expression indicates the read from the state. Writing to a state is done by assigning the state identifier with a value or an expression. For example, the following semantic code segment shows how the states are read and written by an instruction:

```
assign KEYC = sr == 8'd2 ? art[27:0] : KEYC;
assign KEYD = sr == 8'd3 ? art[27:0] : KEYD;
assign DATA = sr == 8'd0 ? {DATA[63:32], art} : {art, DATA[63:32]};
```

The Xtensa™ Instruction Set Architecture (ISA) Reference Manual, Revision 1.0 by Tensilica, Inc. is incorporated herein by reference for the purposes of illustrating examples of instructions that can be implemented within the configurable processor as core instructions and instructions which are available via the selection of configuration options. Further, the Instruction Extension Language (TIE) Reference Manual Revision 1.3, also by Tensilica, Inc., is incorporated by reference to show examples of TIE language instructions which can be used to implement such user-defined instructions.

From the TIE description, new hardware implementing the instructions can be generated using, e.g., a program similar to the one shown in Appendix D. Appendix E shows the code for header files needed to support new instructions as intrinsic functions.

Using the configuration specification, the following can be automatically generated:

- instruction decode logic of the processor 60;
- illegal instruction detection logic for the processor 60;
- the ISA-specific portion of the assembler;
- the ISA-specific support routines for the compiler;
- the ISA-specific portion of the disassembler (used by the debugger); and
- the ISA-specific portion of the simulator.

FIG. 16 is a diagram of how the ISA-specific portions of these software tools are generated. From a user-created TIE description file 400, a TIE parser program 410 generates C code for several programs, each of which produces a file accessed by one or more of the software development tools for information about the user-defined instructions and state. For example, the program `tie2gcc` 420 generates a C header file 470 called `xtensa-tie.h` which contains intrinsic function definitions for new instructions. The program `tie2isa` 430 generates a dynamic linked library (DLL) 480 which contains information on user-defined instruction format (in the Wilson et al. application discussed below, this is effectively a combination of the encode and decode DLLs discussed therein). The program `tie2iss` 440 generates performance modeling routines and produces a DLL 490 containing instruction semantics which, as discussed in the Wilson et al. application, is used by a host compiler to produce a simulator DLL used by the simulator. The program `tie2ver` 450 produces necessary descriptions 500 for user-defined instructions in an appropriate hardware description language. Finally, the program `tie2xtos` 460 produces save and restore code 510 for use by RUR and WUR instructions.

The precise descriptions of instructions and how they access states make it possible to produce efficient logic that can plug into an existing high-performance microprocessor designs. The methods described in connection with this embodiment of the present invention specifically deal with those new instructions which read from or write to one or more state registers. In particular, this embodiment shows how to derive hardware logic for state registers in the context a class of microprocessor implementation styles which all use pipelining as a technique to achieve high performance.

In a pipelined implementation such as the one shown in FIG. 17, a state register is typically duplicated several times, each instantiation representing the value of the state at a particular pipeline stage. In this embodiment, a state is translated into multiple copies of registers consistent with the underlying core processor implementation. Additional bypass and forward logic are also generated, again in a manner consistent with the underlying core processor implementation. For example, to target a core processor implementation that consists of three execution stages, this embodiment would translate a state into three registers connected as shown in FIG. 18. In this implementation, each register 610 - 630 represents the value of the state `in` at one of the three pipeline stages. `ctrl-1`, `ctrl-2`, and `ctrl-3` are control signals used to enable the data latching in the corresponding flip-flops 610- 630.

To make multiple copies of a state register work consistently with the underlying processor implementation requires additional logic and control signals. "Consistently" means that the state should behave exactly the same way as the rest of the processor states under conditions of interrupts, exceptions and pipeline stalls. Typically, a given processor implementation defines certain signals representing various pipeline conditions. Such signals are required to make the pipeline state registers work properly.

In a typical pipelined implementation, the execution unit consists of multiple pipeline stages. The computation of an instruction is carried out in multiple stages in this pipeline. Instruction streams flow through the pipeline in sequence as directed by the control logic. At any given time, there can be up to n instructions being executed in the pipeline, where n is the number of stages. In a superscalar processor, also implementable using the present invention, the number of instructions in the pipeline can be n•w, wherein w is the issue width of the processor.

The role of the control logic is to make sure the dependencies between the instructions are obeyed and any interference between instructions is resolved. If an instruction uses data computed by an earlier instruction, special hardware is needed to forward the data to the later instruction without stalling the pipeline. If an interrupt occurred, all instructions in the pipeline need to be killed and later on re-executed. When an instruction cannot be executed because its input data or the computational hardware it needs is not available, the instruction must be stalled. One cost effective way of stalling an instruction is to kill it in its first execution stage and re-execute the instruction in the next cycle. A consequence of this technique is creating an invalid stage (bubble) in the pipeline. This bubble flows through the pipeline

along with other instructions. At the end of the pipeline where instructions are committed, the bubbles are thrown away.

Using the above three-stage pipeline example, a typical implementation of such a processor state requires the additional logic and connections shown in FIG 19.

Under normal situations, a value computed in a stage will be forwarded to the next instructions immediately without waiting for the value to reach the end of the pipeline in order to reduce the number of pipeline stalls introduced by the data dependencies. This is accomplished by sending the output of the first flip-flop 610 directly to the semantic block such that it can be used immediately by the next instruction. To handle abnormal conditions such as interrupts and exceptions, the implementation requires the following control signals: `Kill_1`, `Kill_all` and `Valid_3`.

Signal "`Kill_1`" indicates that the instruction currently in the first pipeline stage 110 must be killed due to reasons such as not having the data it needs to proceeds. Once the instruction is killed, it will be retried in the next cycle. Signal "`Kill_all`" indicates that all the instructions currently in the pipeline must be killed for reasons such as an instruction ahead of them has generated an exception or an interrupt has occurred. Signal "`Valid_3`" indicates whether the instruction currently in the last stage 630 is valid or not. Such a condition is often the result of killing an instruction in the first pipeline stage 610 and causing a bubble (invalid instruction) in the pipeline. "`Valid_3`" simply indicates whether the instruction in the third pipeline stage is valid or a bubble. Clearly, only valid instructions should be latched.

FIG. 20 shows the additional logic and connections needed to implement the state register. It also shows how to construct the control logic to drive the signals "`ctrl-1`", "`ctrl-2`", and "`ctrl-3`" such that this state-register implementation meets the above requirements. The following is sample HDL code automatically generated to implement the state register as shown in FIG. 19.

```
module tie_enflop(tie_out, tie_in, en, clk);
parameter size = 32;
output [size-1:0] tie_out;
input [size-1:0]  tie_in;
input             en;
input             clk;
reg [size-1:0]    tmp;
assign tie_out = tmp;
always @(posedge clk) begin
  if (en)
    tmp <= #1 tie_in;
end
endmodule

module tie_athens_state(ns, we, ke, kp, vw, clk, ps);
parameter size = 32;
input [size-1:0] ns;  // next state
input we;        // write enable
input ke;        // Kill E state
input kp;        // Kill Pipeline
```



```
input vw;         // Valid W state
input clk;        // clock
output [size-1:0] ps; // present state

wire [size-1:0] se;   // state at E stage
wire [size-1:0] sm;   // state at M stage
wire [size-1:0] sw;   // state at W stage
wire [size-1:0] sx;   // state at X stage
wire ee;          // write enable for EM register
wire ew;          // write enable for WX register

assign se = kp ? sx : ns;
assign ee = kp | we & ~ke;
assign ew = vw & ~kp;
assign ps = sm;

tie_enflop #(size) state_EM(.tie_out(sm), .tie_in(se), .en(ee),
    \ .clk(clk));
tie_enflop                                                #(size)
    state_MW(.tie_out(sw), .tie_in(sm), .en(1'b1), \ .clk(clk));
tie_enflop #(size) state_WX(.tie_out(sx), .tie_in(sw), .en(ew),
    \ .clk(clk));

endmodule
```

Using the above pipelined state register model, the present state value of the state is passed to the semantic block as an input variable if the semantic block specifies the state as its input. If the semantic block has the logic to generate the new value for a state, an output signal is created. This output signal is used as the next-state input to the pipelined state register.

This embodiment allows multiple semantic description blocks each of which describes the behavior for multiple instructions. Under this unrestricted description style, it is possible that only a subset of the semantic blocks produce next-state output for a given state. Furthermore, it is also possible that a given semantic block produces the next-state output conditionally depending on what instruction it is executing at a given time. Consequently, additional hardware logic is needed to combine the next-state outputs from all semantic blocks to form the input to the pipelined state register. In this embodiment of the invention, a signal is automatically derived for each semantic block indicating whether this block has produced a new value for the state. In another embodiment, such a signal can be left to the designer to specify.

FIG. 20 shows how to combine the next-state output of a state from several semantic blocks `s1` - `sn` and appropriately select one to input to the state register. In this Figure, `op1_1` and `op1_2` are opcode signals for the first semantic block, `op2_1` and `op2_2` are opcode signals for the second semantic block, etc. The next-state output of semantic block `i` is `si` (there are multiple next-state outputs for the block if there are multiple state registers). The signal indicating that semantic block `i` has produced a new value for the state is `si_we`. Signal `s_we` indicates whether any of the semantic blocks

produce a new value for the state, and is used as an input to the pipelined state register as the write-enable signal.

Even though the expressive power of the multiple semantic block is no more than that of a single one, it does provide a way for implementing more structured descriptions, typically by grouping related instructions into a single block. Multiple semantic blocks can also lead to simpler analysis of instructions effects because of the more restricted scope in which the instructions are implemented. On the other hand, there are often reasons for a single semantic block to describe the behavior of multiple instructions. Most often, it is because the hardware implementation of these instructions share common logic. Describing multiple instructions in a single semantic block usually leads to more efficient hardware design hardware design.

Because of interrupts and exceptions, it is necessary for software to restore and load the values of the states to and from data memory. Based on the formal description of the new states and new instructions, it is possible to automatically generate such restore and load instructions. In an embodiment of the invention, the logic for the restore and load instructions is automatically generated as two semantic blocks which can then be recursively translated into actual hardware just like any other blocks. For example, from the following declaration of states:

```
state [63:0] DATA cpn=0 autopack
state [27:0] KEYC cpn=1 nopack
state [27:0] KEYD cpn=1
user_register 0 = DATA[31:0];
user_register 1 = DATA[63:32];
user_register 2 = KEYC;
user_register 3 = KEYD;
```

the following semantic block can be generated to read the values of "DATA", "KEYC", and "KEYD" into general purpose registers:

```
iclass rur (RUR) {out arr, in st} {in DATA, in KEYC, in KEYD}
semantic rur (RUR) {
    wire sel_0 = (st == 8'd0);
    wire sel_1 = (st == 8'd1);
    wire sel_2 = (st == 8'd2);
    wire sel_3 = (st == 8'd3);
    assign arr = {32{sel_0}} & DATA[31:0]  |
                 {32{sel_1}} & DATA[64:32] |
                 {32{sel_2}} & KEYC        |
                 {32{sel_3}} & KEYD;
}
```

FIG. 21 shows the block diagram of the logic corresponding to this kind of semantic logic. The input signal "st" is compared with various constants to form various selection signals which are used to select certain bits from the state registers in a way consistent with the user_register specification. Using the previous state declaration, bit 32 of DATA maps to bit 0 of the second user register. Therefore, the second input of the MUX in this diagram should be connected to the 32nd bit of the DATA state.



The following semantic block can be generated to write the states "DATA", "KEYC", and "KEYD" with values from general purpose registers

```
iclass wur (WUR) {in art, in sr} {out DATA. out KEYC, out KEYD}
semantic wur (WUR) {
    wire sel_0 = (st == 8'd0);
    wire sel_1 = (st == 8'd1);
    wire sel_2 = (st == 8'd2);
    wire sel_3 = (st == 8'd3);
    assign DATA = {sel_1 ? art : DATA[63:32], sel_0 ? art : DATA[31:0]};
    assign KEYC = art;
    assign KEYD = art;
    assign DATA_we = WUR;
    assign KEYC_we = WUR & sel_2;
    assign KEYD_we = WUR & sel_3;
}
```

FIG. 22 shows the logic for the jth bit of state S when it is mapped to the kth bit of the ith user register. If the user_register number "st" in a WUR instruction is "i", the kth bit of "ars" is loaded into the S[j] register; otherwise, the original value of S[j] is recirculated. In addition, if any bit of the state S is reloaded, the signal S_we is enabled.

The TIE user_register declaration specifies a mapping from additional processor state defined by state declarations to an identifier used by these RUR and WUR instructions to read and write this state independent of the TIE instructions.

Appendix F shows the code for generating RUR and WUR instructions.

The primary purpose for RUR and WUR is for task switching. In a multi-tasking environment, the multiple software tasks share the processor, running according to some scheduling algorithm. When active, the task's state resides in the processor registers. When the scheduling algorithm decides to switch to another task, the state held in the processor registers is saved to memory, and another task's state is loaded from memory to the processor registers. The Xtensa™ Instruction Set Architecture (ISA) includes the RSR and WSR instructions to read and write the state defined by the ISA. For example, the following code is part of the task "save to memory":

```
    // save special registers
    rsr  a0, SAR
    rsr  a1, LCOUNT
    s32i a0, a3, UEXCSAVE +  0
    s32i a1, a3, UEXCSAVE +  4
    rsr  a0, LBEG
    rsr  a1, LEND
    s32i a0, a3, UEXCSAVE +  8
    s32i a1, a3, UEXCSAVE + 12
;if (config_get_value("IsaUseMAC16") ) {
    rsr  a0, ACCLO
    rsr  a1, ACCHI
    s32i a0, a3, UEXCSAVE + 16
    s32i a1, a3, UEXCSAVE + 20
    rsr  a0, MR_0
```

```
    rsr  a1, MR_1
    s32i a0, a3, UEXCSAVE + 24
    s32i a1, a3, UEXCSAVE + 28
    rsr  a0, MR_2
    rsr  a1, MR_3
    s32i a0, a3, UEXCSAVE + 32
    s32i a1, a3, UEXCSAVE + 36
;}
```

and the following code is part of the task "restore from memory":

```
    // restore special registers
    l32i a2, a1, UEXCSAVE +  0
    l32i a3, a1, UEXCSAVE +  4
    wsr  a2, SAR
    wsr  a3, LCOUNT
    l32i a2, a1, UEXCSAVE +  8
    l32i a3, a1, UEXCSAVE + 12
    wsr  a2, LBEG
    wsr  a3, LEND
;if (config_get_value("IsaUseMAC16") ) {
    l32i a2, a1, UEXCSAVE + 16
    l32i a3, a1, UEXCSAVE + 20
    wsr  a2, ACCLO
    wsr  a3, ACCHI
    l32i a2, a1, UEXCSAVE + 24
    l32i a3, a1, UEXCSAVE + 28
    wsr  a2, MR_0
    wsr  a3, MR_1
    l32i a2, a1, UEXCSAVE + 32
    l32i a3, a1, UEXCSAVE + 36
    wsr  a2, MR_2
    wsr  a3, MR_3
;}
```

where SAR, LCOUNT, LBEG, LEND are processor state registers part of the core Xtensa™ ISA, and ACCLO, ACCHI, MR_0, MR_1, MR_2, and MR_3 are part of the MAC16 Xtensa™ ISA option. (The registers are saved and restored in pairs to avoid pipeline interlocks.)

When the designer defines new state with TIE, it too must be task switched like the state above. One possibility would be for the designer to simply go edit the task switch code (a portion of which is given above) and add RUR/S32I and L32I/WUR instructions analogous to the code above. However, configurable processors are most effective when the software is generated automatically and is correct by construction. Thus this invention includes a facility to augment the task switch code automatically. The following tpp lines are added to the above save task:

```
    ; my $off = 0;
    ; my $i;
    ; for ($i = 0; $i < $#user_registers; $i += 2) {
        rur   a2, `$user_registers[$i+0]`
        rur   a3, `$user_registers[$i+1]`
        s32i  a2, UEXCUREG + `$off + 0`
        s32i  a3, UEXCUREG + `$off + 4`
    ;   $off += 8;
    ; }
    ; if (@user_registers & 1) {
    ;   # odd number of user registers
        rur   a2, `$user_registers[$#user_registers]`
        s32i  a2, UEXCUREG + `$off + 0`
    ;   $off += 4;
    ; }
```

and the following lines are added to the above restore task:

```
    ; my $off = 0;
    ; my $i;
    ; for ($i = 0; $i < $#user_registers; $i += 2) {
        l32i  a2, UEXCUREG + `$off + 0`
        l32i  a3, UEXCUREG + `$off + 4`
        wur   a2, `$user_registers[$i+0]`
        wur   a3, `$user_registers[$i+1]`
    ;   $off += 8;
    ; }
    ; if (@user_registers & 1) {
    ;   # odd number of user registers
        l32i  a2, UEXCUREG + `$off + 0`
        wur   a2, `$user_registers[$#user_registers]`
    ;   $off += 4;
    ; }
```

Finally, the task state area in memory must have additional space allocated for the user register storage, and the offset of this space from the base of the task save pointer is defined as the assembler constant UEXCUREG. This save area was previously defined by the following code

```
    #define UEXCREGSIZE (16*4)
    #define UEXCPARMSIZE (4*4)
    ;if (&config_get_value("IsaUseMAC16") ) {
    #define UEXCSAVESIZE (10*4)
    ;} else {
    #define UEXCSAVESIZE (4*4)
    ;}
    #define UEXCMISCSIZE (2*4)
    #define UEXCPARM 0
    #define UEXCREG (UEXCPARM+UEXCPARMSIZE)
    #define UEXCSAVE (UEXCREG+UEXCREGSIZE)
    #define UEXCMISC (UEXCSAVE+UEXCSAVESIZE)
    #define UEXCWIN (UEXCMISC+0)
    #define                                                UEXCFRAME
(UEXCREGSIZE+UEXCPARMSIZE+UEXCSAVESIZE+UEXCMISCSIZE)
```

which is changed to

```
    #define UEXCREGSIZE (16*4)
    #define UEXCPARMSIZE (4*4)
    ;if (&config_get_value("IsaUseMAC16") ) {
    #define UEXCSAVESIZE (10*4)
    ;} else {
    #define UEXCSAVESIZE (4*4)
```

```
;}
#define UEXCMISCSIZE (2*4)
#define UEXCUREGSIZE `@user_registers * 4`
#define UEXCPARM 0
#define UEXCREG (UEXCPARM+UEXCPARMSIZE)
#define UEXCSAVE (UEXCREG+UEXCREGSIZE)
#define UEXCMISC (UEXCSAVE+UEXCSAVESIZE)
#define UEXCUREG (UEXCMISC+UEXCMISCSIZE)
#define UEXCWIN (UEXCUREG+0)
#define UEXCFRAME \

    (UEXCREGSIZE+UEXCPARMSIZE+UEXCSAVESIZE+UEXCMISCSIZE+UEXCUREGSIZE)
```

This code is dependent on there being a tpp variable `@user_registers` with a list of the user register numbers. This is simply a list created from the first argument of every `user_register` statement.

In some more complex microprocessor implementations, a state can be computed in different pipeline states. Handling this requires several extensions (albeit simple ones) to the process described here. First, the specification language needs to be extended to be able to associate a semantic block with a pipeline stage. This can be accomplished in one of several ways. In one embodiment, the associated pipeline stage can be specified explicitly with each semantic block. In another embodiment, a range of pipeline stages can be specified for each semantic block. In yet another embodiment, the pipeline stage for a given semantic block can be automatically derived depending on the required computational delay.

The second task in supporting state generation at different pipeline stages is to handle interrupts, exceptions, and stalls. This usually involves adding appropriate bypass and forward logic under the control of pipeline control signals. In one embodiment, a generate-usage diagram can be generated to indicate the relationship between when the state is generated and when it is used. Based on application analysis, appropriate forward logic can be implemented to handle the common situation and interlock logic can be generated to stall the pipeline for the cases not handled by the forwarding logic.

The method for modifying the instruction issue logic of the base processor dependent on the algorithms employed by the base processor. However, generally speaking, instruction issue logic for most processors, whether single-issue or superscalar, whether for single-cycle or multi-cycle instructions, depends only on for the instruction being tested for issue:

1. signals that indicate for each processor state element whether the instruction uses the states as a source;

2. signals that indicate for each processor state element whether the instruction uses the states as a destination; and

3. signals that indicate for each functional unit whether the instruction uses the functional units;

These signals are used to perform issue to pipeline and cross-issue checks and to update the pipeline status in the pipeline-dependent issue logic. TIE contains all the necessary information to augment the signals and their equations for the new instructions.



First, each TIE state declaration cause a new signal to be created for the instruction issue logic. Each `in` or `inout` operand or state listed in the third or fourth argument to the `iclass` declaration adds the instruction decode signal for the instructions listed in the second argument to the first set of equations for the specified processor state element.

Second, each `out` or `inout` operand or state listed in the third or fourth argument to the `iclass` declaration adds the instruction decode signal for the instructions listed in the second argument to the second set of equations for the specified processor state element.

Third, the logic created from each TIE semantic blocks represents a new functional unit, so a new unit signals are created, and the decode signals for the TIE instructions specified for the semantic block are OR'd together to form the third set of equations.

When an instruction is issued, the pipeline status must be updated for future issue decisions. Again the method for modifying the instruction issue logic of the base processor dependent on the algorithms employed by the base processor. However, again some general observations are possible. The pipeline status must provide the following status back to the issue logic:

4. signals that indicate for each issued instruction destination when that result is available for bypass;

5. signals for each functional unit that indicate the functional unit is ready for another instruction.

The embodiment described herein is a single-issue processor where the designer-defined instructions are limited to a single cycle of logic computation. In this case the above simplifies considerably. There is no need for the functional unit checks or cross-issue checks, and no single-cycle instruction can make a processor state element to be not pipeready for the next instruction. Thus the issue equation becomes just

```
issue = (~src1use | src1pipeready) & (~src2use | src2pipeready)
        .
        .
        .
        & (~srcNuse | srcNpipeready);
```

and where the src[i]pipeready signals are unaffected by the additional instructions and src[i]use are the first set of equations described and modified as explained above. In this embodiment, the fourth and fifth set of signals are not required. For an alternate embodiment that is multi-issue with multi-cycle, the TIE specification would be augmented with a latency specification for each instruction giving the number of cycles over which to pipeline the computation.

The fourth set of signals would be generated in each semantic block pipe stage by OR'ing together the instruction decode signals for each instruction that completes in that stage according to the specification.

By default the generated logic will be fully pipelined, and so the TIE generated functional units will always be ready one cycle after accepting an instruction. In this case the fifth set of signals for TIE semantic blocks is always asserted. When it is necessary to reuse logic in the semantic blocks over

multiple cycles, a further specification will specify how many cycles the functional unit will be in use by such instructions. In this case the fifth set of signals would be generated in each semantic block pipe stage by OR'ing together the instruction decode signals for each instruction that finishes with the specified cycle count in that stage.

Alternatively, in a still different embodiment, it may be left as an extension to TIE for the designer to specify the result ready and functional unit ready signals.

Examples of code processed according to this embodiment are shown in the attached Appendices. For brevity, these will not be explained in detail; however, they will be readily understood by those skilled in the art after review of the reference manuals described above. Appendix G is an example of implementation of an instruction using the TIE language; Appendix H shows what the TIE compiler generates for the compiler using such code. Similarly, Appendix I shows what the TIE compiler generates for the simulator; Appendix J shows what the TIE compiler generates for macro expanding the TIE instructions in a user application; Appendix K shows what tie compiler generates to simulate TIE instructions in native mode; Appendix L shows what tie compiler generates as Verilog HDL description for the additional hardware; and Appendix M shows what the TIE compiler generates as Design Compiler script to optimize the Verilog HDL description above to estimate the area and speed impact of the TIE instruction on the total CPU size and performance.

As noted above, to begin the processor configuration procedure, a user begins by selecting a base processor configuration via the GUI described above. As part of the process, a software development system 30 is built and delivered to the user as shown in FIG. 1. The software development system 30 contains four key components relevant to another aspect of the present invention, shown in greater detail in FIG. 6: a compiler 108, an assembler 110, an instruction set simulator 112 and a debugger 130.

A compiler, as is well known to those versed in the art, converts user applications written in high level programming languages such as C or C++ into processor-specific assembly language. High level programming languages such as C or C++ are designed to allow application writers to describe their application in a form that is easy for them to precisely describe. These are not languages understood by processors. The application writer need not necessarily worry about all the specific characteristics of the processor that will be used. The same C or C++ program can typically be used with little or no modification on many different types of processors.

The compiler translates the C or C++ program into assembly language. Assembly language is much closer to machine language, the language directly supported by the processor. Different types of processors will have their own assembly language. Each assembly instruction often directly represents one machine instruction, but the two are not necessarily identical. Assembly instructions are designed to be human readable strings. Each instruction and operand is given a meaningful name or mnemonic, allowing humans to read assembly instructions and easily understand what operations will be performed by the machine. Assemblers convert from assembly language into machine language. Each assembly



instruction string is efficiently encoded by the assembler into one or more machine instructions that can be directly and efficiently executed by the processor.

Machine code can be directly run on the processor, but physical processors are not always immediately available. Building physical processors is a time-consuming and expensive process. When selecting potential processor configurations, a user cannot build a physical processor for each potential choice. Instead, the user is provided with a software program called a simulator. The simulator, a program running on a generic computer, is able to simulate the effects of running the user application on the user configured processor. The simulator is able to mimic the semantics of the simulated processor and is able to tell the user how quickly the real processor will be able to run the user's application.

A debugger is a tool that allows users to interactively find problems with their software. The debugger allows users to interactively run their programs. The user can stop the program's execution at any time and look at her C source code, the resultant assembly or machine code. The user can also examine or modify the values of any or all of her variables or the hardware registers at a break point. The user can then continue execution -- perhaps one statement at a time, perhaps one machine instruction at a time, perhaps to a new user-selected break point.

All four components 108, 110, 112 and 130 need to be aware of user-defined instructions 750 (see FIG. 3) and the simulator 112 and debugger 130 must additionally be aware of user-defined state 752. The system allows the user to access user-defined instructions 750 via intrinsics added to user C and C++ applications. The compiler 108 must translate the intrinsic calls into the assembly language instructions 738 for the user-defined instructions 750. The assembler 110 must take the new assembly language instructions 738, whether written directly by the user or translated by the compiler 108, and encode them into the machine instructions 740 corresponding to the user-defined instructions 750. The simulator 112 must decode the user-defined machine instructions 740. It must model the semantics of the instructions, and it must model the performance of the instructions on the configured processor. The simulator 112 must also model the values and performance implications of user-defined state. The debugger 130 must allow the user to print the assembly language instructions 738 including user-defined instructions 750. It must allow the user to examined and modify the value of user-defined state.

In this aspect of the present invention, the user invokes a tool, the TIE compiler 702, to process the current potential user-defined enhancements 736. The TIE compiler 702 is different from the compiler 708 that translates the user application into assembly language 738. The TIE compiler 702 builds components which enable the already-built base software system 30 (compiler 708, assembler 710 and simulator 712 and debugger 730) to use the new, user-defined enhancements 736. Each element of the software system 30 uses a somewhat different set of components.

FIG. 24 is a diagram of how the TIE-specific portions of these software tools are generated. From the user-defined extension file 736, the TIE compiler 702 generates C code for several programs, each of which produces a file accessed by one or more of the software development tools for information about the user-defined instructions and state. For example, the program `tie2gcc` 800 generates a C header file

842 called `xtensa-tie.h` (described in greater detail below) which contains intrinsic function definitions for new instructions. The program `tie2isa` 810 generates a dynamic linked library (DLL) 844/848 which contains information on user-defined instruction format (a combination of encode DLL 844 and decode DLL 848 described in greater detail below). The program `tie2iss` 840 generates C code 870 for performance modeling and instruction semantics which, as discussed below, is used by a host compiler 846 to produce a simulator DLL 849 used by the simulator 712 as described in greater detail below. The program `tie2ver` 850 produces necessary descriptions 850 for user-defined instructions in an appropriate hardware description language. Finally, the program `tie2xtos` 860 produces save and restore code 810 to save and restore the user-defined state for context switching. Additional information on the implementation of user-defined state can be found in the afore-mentioned Wang et al. application.

Compiler 708

In this embodiment, the compiler 708 translates intrinsic calls in the user's application into assembly language instructions 738 for the user-defined enhancements 736. The compiler 708 implements this mechanism on top of the macro and inline assembly mechanisms found in standard compilers such as the GNU compilers. For more information on these mechanisms, see, e.g., GNU C and C++ Compiler User's Guide, EGCS Version 1.0.3.

Consider a user who wishes to create a new instruction `foo` that operates on two registers and returns a result in a third register. The user puts the instruction description in a user-defined instruction file 750 in a particular directory and invokes the TIE compiler 702. The TIE compiler 702 creates a file 742 with a standard name such as `xtensa-tie.h`. That file contains the following definition of `foo`.

```
#define foo(ars, art) \
      ({int arr; asm volatile("foo %0,%1,%2" : "=a" (arr) : \
      "a" (ars), "a" (art)); })
```

When the user invokes the compiler 708 on her application, she tells the compiler 708 either via a command line option or an environment variable the name of the directory with the user-defined enhancements 736. That directory also contains the `xtensa-tie.h` file 742. The compiler 708 automatically includes the file `xtensa-tie.h` into the user C or C++ application program being compiled as if the user had written the definition of `foo` herself. The user has included intrinsic calls to the instruction foo in her application. Because of the included definition, the compiler 708 treats those intrinsic calls as calls to the included definition. Based on the standard macro mechanism provided by the compiler 708, the compiler 708 treats the call to the macro foo as if the user had directly written the assembly language statement 738 rather than the macro call. That is, based on the standard inline assembly mechanism, the compiler 708 translates the call into the single assembly instruction foo. For example, the user might have a function that contains a call to the intrinsic `foo`:

```
int fred(int a, int b)
```



```
{
return foo(a,b);
}
```

The compiler translates the function into the following assembly language subroutine using the user defined instruction `foo`:

```
fred:
.frame sp, 32
entry sp, 32
#APP
foo a2,a2,a3
#NO_APP
retw.n
```

When the user creates a new set of user-defined enhancements 736, no new compiler needs to be rebuilt. The TIE compiler 702 merely creates the file `xtensa-tie.h` 742 which is automatically included by the prebuilt compiler 708 into the user's application.

Assembler 710

In this embodiment, the assembler 710 uses an encode library 744 to encode assembly instructions 750. The interface to this library 744 includes functions to:

-- translate an opcode mnemonic string to an internal opcode representation;

-- provide the bit patterns to be generated for each opcode for the opcode fields in a machine instruction 740; and

-- encode the operand value for each instruction operand and insert the encoded operand bit patterns into the operand fields of a machine instruction 740.

As an example, consider our previous example of a user function that calls the intrinsic `foo`. The assembler might take the "`foo a2, a2, a3`" instruction and convert it into the machine instruction represented by the hexadecimal number 0x62230, where the high order 6 and the lower order 0 together represent the opcode for `foo`, and the 2, 2 and 3 represent the three registers a2, a2 and a3 respectively.

The internal implementations of these functions are based on a combination of tables and internal functions. Tables are easily generated by the TIE compiler 702, but their expressiveness is limited. When more flexibility is needed, such as when expressing the operand encoding functions, the TIE compiler 702 can generate arbitrary C code to be included in the library 744.

Consider again the example of "`foo a2, a2, a3`". Every register field is simply encoded with the number of the register. The TIE compiler 702 creates the following function that checks for legal register values, and if the value is legal, returns the register number:

```
xtensa_encode_result encode_r (valp)
u_int32_t *valp;
{
      u_int32_t val = *valp;
      if ((val >> 4) != 0)
```

```
        return xtensa_encode_result_too_high;
        *valp = val;
        return xtensa_encode_result_ok;
    }
```

If all encodings were so simple, no encoding functions would be needed; a table would suffice. However, the user is allowed to choose more complicated encodings. The following encoding, described in the TIE language, encodes every operand with a number that is the value of the operand divided by 1024. Such an encoding is useful to densely encode values that are required to be multiples of 1024.

```
        operand tx10 t {t << 10} { tx10 >> 10 }
```

The TIE compiler converts the operand encoding description into the following C function.

```
    xtensa_encode_result encode_tx10 (valp)
    u_int32_t *valp;
    {
        u_int32_t t, tx10;
        tx10 = *valp;
        t = (tx10 >> 10) & 0xf;
        tx10 = decode_tx10(t);
        if (tx10 != *valp) {
        return xtensa_encode_result_not_ok;
    } else {
        *valp = t;
    }
    return xtensa_encode_result_ok;
    }
```

A table can not be used for such an encoding since the domain of possible values for the operand is very large. A table would have to be very large.

In an embodiment of the encode library 744, one table maps opcode mnemonic strings to the internal opcode representation. For efficiency, this table may be sorted or it may be a hash table or some other data structure allowing efficient searching. Another table maps each opcode to a template of a machine instruction with the opcode fields initialized to the appropriate bit patterns for that opcode. Opcodes with the same operand fields and operand encodings are grouped together. For each operand in one of these groups, the library contains a function to encode the operand value into a bit pattern and another function to insert those bits into the appropriate fields in a machine instruction. A separate internal table maps each instruction operand to these functions. Consider an example where the result register number is encoded into bits 12..15 of the instruction. The TIE compiler 702 will generate the following function that sets bits 12..15 of the instruction with the value (number) of the result register:

```
    void set_r_field (insn, val)
    xtensa_insnbuf insn;
    u_int32_t val;
    {
        insn[0] = (insn[0] & 0xffff0fff) | ((val << 12) & 0xf000);
    }
```



To allow changing user-defined instructions without rebuilding the assembler 710, the encode library 744 is implemented as a dynamically linked library (DLL). DLLs are a standard way to allow a program to extend its functionality dynamically. The details of handling DLLs vary across different host operating systems, but the basic concept is the same. The DLL is dynamically loaded into a running program as an extension of the program's code. A run-time linker resolves symbolic references between the DLL and the main program and between the DLL and other DLLs already loaded. In the case of the encode library or DLL 744, a small portion of the code is statically linked into the assembler 710. This code is responsible for loading the DLL, combining the information in the DLL with the existing encode information for the pre-built instruction set 746 (which may have been loaded from a separate DLL), and making that information accessible via the interface functions described above.

When the user creates new enhancements 736, she invokes the TIE compiler 702 on a description of the enhancements 736. The TIE compiler 702 generates C code defining the internal tables and functions which implement the encode DLL 744. The TIE compiler 702 then invokes the host system's native compiler 746 (which compiles code to run on the host rather than on the processor being configured) to create the encode DLL 144 for the user-defined instructions 750. The user invokes the pre-built assembler 710 on her application with a flag or environment variable pointing to the directory containing the user-defined enhancements 736. The prebuilt assembler 710 dynamically opens the DLL 744 in the directory. For each assembly instruction, the prebuilt assembler 710 uses the encode DLL 744 to look up the opcode mnemonic, find the bit patterns for the opcode fields in the machine instruction, and encode each of the instruction operands.

For example, when the assembler 710 sees the TIE instruction "`foo a2, a2, a3`", the assembler 710 sees from a table that the "`foo`" opcode translates into the number 6 in bit positions 16 to 23. From a table, it finds the encoding functions for each of the registers. The functions encode a2 into the number 2, the other a2 into the number 2 and a3 into the number 3. From a table, it finds the appropriate set functions. `Set_r_field` puts the result value 2 into bit positions 12..15 of the instruction. Similar set functions appropriately place the other 2 and the 3.

Simulator 712

The simulator 712 interacts with user-defined enhancements 736 in several ways. Given a machine instruction 740, the simulator 712 must decode the instruction; i.e., break up the instruction into the component opcode and operands. Decoding of user-defined enhancements 736 is done via a function in a decode DLL 748 (it is possible that the encode DLL 744 and the decode DLL 748 are actually a single DLL). For example, consider a case where the user defines three opcodes; `foo1`, `foo2` and `foo3` with encodings 0x6, 0x16 and 0x26 respectively in bits 16 to 23 of the instruction and with 0 in bits 0 to 3. The TIE compiler 702 generates the following decode function that compares the opcode with the opcodes of all the user-defined instructions 750:

```
int decode_insn(const xtensa_insnbuf insn)
{
     if ((insn[0] & 0xff000f) == 0x60000) return xtensa_foo1_op;
     if ((insn[0] & 0xff000f) == 0x160000) return xtensa_foo2_op;
     if ((insn[0] & 0xff000f) == 0x260000) return xtensa_foo3_op;
     return XTENSA_UNDEFINED;
}
```

With a large number of user-defined instructions, comparing an opcode against all possible user-defined instructions 750 can be expensive, so the TIE compiler can instead use a hierarchical set of switch statements

```
switch (get_op0_field(insn)) {
  case 0x0:
    switch (get_op1_field(insn)) {
      case 0x6:
        switch (get_op2_field(insn)) {
          case 0x0: return xtensa_foo1_op;
          case 0x1: return xtensa_foo2_op;
          case 0x2: return xtensa_foo3_op;
          default: return XTENSA_UNDEFINED;
        }
      default: return XTENSA_UNDEFINED;
    }
  default: return XTENSA_UNDEFINED;
}
```

In addition to decoding instruction opcodes, the decode DLL 748 includes functions for decoding instruction operands. This is done in the same manner as for encoding operands in the encode DLL 744. First, the decode DLL 748 provides functions to extract the operand fields from machine instructions. Continuing the previous examples, the TIE compiler 702 generates the following function to extract a value from bits 12 to 15 of an instruction:

```
u_int32_t get_r_field (insn)
     xtensa_insnbuf insn;
{
  return ((insn[0] & 0xf000) >> 12);
}
```

The TIE description of an operand includes specifications of both encoding and decoding, so whereas the encode DLL 744 uses the operand encode specification, the decode DLL 748 uses the operand decode specification. For example, the TIE operand specification:

```
operand tx10 t {t << 10} {tx10 >> 10}
```

produces the following operand decode function:

```
u_int32_t decode_tx10 (val)
    u_int32_t val;
{
  u_int32_t t, tx10;
  t = val;
```



```
  tx10 = t << 10;
  return tx10;
}
```

When the user invokes the simulator 712, she tells the simulator 712 the directory containing the decode DLL 748 for the user-defined enhancements 736. The simulator 712 opens the appropriate DLL. Whenever the simulator 712 decodes an instruction, if that instruction is not successfully decoded by the decode function for the pre-built instruction set, the simulator 712 invokes the decode function in the DLL 748.

Given a decoded instruction 750, the simulator 712 must interpret and model the semantics of the instruction 750. This is done functionally. Every instruction 750 has a corresponding function that allows the simulator 712 to model the semantics of that instruction 750. The simulator 712 internally keeps track of all states of the simulated processor. The simulator 712 has a fixed interface to update or query the processor's state. As noted above, user-defined enhancements 736 are written in the TIE hardware description language which is a subset of Verilog. The TIE compiler 702 converts the hardware description into a C function used by the simulator 712 to model the new enhancements 736. Operators in the hardware description language are translated directly into the corresponding C operators. Operations that read state or write state are translated into the simulator's interface to update or query the processor's state.

As an example in this embodiment, consider a user creating an instruction 750 to add two registers. This example is chosen for simplicity. In the hardware description language, the user might describe the semantics of the add as follows:

```
semantic add { add } { assign arr = ars + art; }
```

The output register, signified by the built-in name `arr`, is assigned the sum of the two input registers, signified by the built in names `ars` and `art`. The TIE compiler 702 takes this description and generates a semantic function used by the simulator 712:

```
void add_func(u32 _OPND0_, u32 _OPND1_, u32 _OPND2_, u32 _OPND3_)
{
set_ar( _OPND0_, ar( _OPND1_ ) + ar( _OPND2_ ) );
pc_incr( 3 );
}
```

The hardware operator "+" is translated directly into the C operator "+". The reads of the hardware registers `ars` and `art` are translated into a call of the simulator 712 function call "`ar`". The write of the hardware register `arr` is translated into a call to the simulator 712 function "`set_ar`". Since every instruction implicitly increments the program counter, `pc`, by the size of the instruction, the TIE compiler 702 also generates a call to the simulator 712 function that increments the simulated `pc` by 3, the size of the add instruction.

When the TIE compiler 702 is invoked, it creates semantic functions as described above for every user-defined instruction. It also creates a table that maps all the opcode names to the associated semantic functions. The table and functions are compiled using the standard compiler 746 into the simulator DLL 749. When the user invokes the simulator 712, she tells the simulator 712 the directory containing the user-defined enhancements 736. The simulator 712 opens the appropriate DLL. Whenever the simulator 712 is invoked, it decodes all the instructions in the program and creates a table that maps instructions to the associated semantic functions. When creating the mapping, the simulator 712 opens the DLL and searches for the appropriate semantic functions. When simulating the semantics of a user-defined instruction 736, the simulator 712 directly invokes the function in the DLL.

In order to tell the user how long an application would take to run on the simulated hardware, the simulator 712 needs to simulate the performance effects of an instruction 750. The simulator 712 uses a pipeline model for this purpose. Every instruction executes over several cycles. In each cycle, an instruction uses different resources of the machine. The simulator 712 begins trying to execute all the instructions in parallel. If multiple instructions try to use the same resource in the same cycle, the latter instruction is stalled waiting for the resource to free. If a latter instruction reads some state that is written by an earlier instruction but in a later cycle, the latter instruction is stalled waiting for the value to be written. The simulator 712 uses a functional interface to model the performance of each instruction. A function is created for every type of instruction. That function contains calls to the simulator's interface that models the performance of the processor.

For example, consider a simple three register instruction `foo`. The TIE compiler might create the following simulator function:

```
void foo_sched (u32 op0, u32 op1, u32 op2, u32 op3)
{
     pipe_use_ifetch (3);
     pipe_use (REGF32_AR, op1, 1);
     pipe_use (REGF32_AR, op2, 1);
     pipe_def (REGF32_AR, op0, 2);
     pipe_def_ifetch (-1);
}
```

The call to `pipe_use_ifetch` tells the simulator 712 that the instruction will require 3 bytes to be fetched. The two calls to `pipe_use` tell the simulator 712 that the two input registers will be read in cycle 1. The call to `pipe_def` tells the simulator 712 that the output register will be written in cycle 2. The call to `pipe_def_ifetch` tells the simulator 712 that this instruction is not a branch, hence the next instruction can be fetched in the next cycle.

Pointers to these functions are placed in the same table as the semantic functions. The functions themselves are compiled into the same DLL 749 as the semantic functions. When the simulator 712 is invoked, it creates a mapping between instructions and performance functions. When creating the mapping, the simulator 712 opens the DLL 749 and searches for the appropriate performance functions.



When simulating the performance of a user-defined instruction 736, the simulator 712 directly invokes the function in the DLL 749.

Debugger 730

The debugger interacts with user-defined enhancements 750 in two ways. First, the user has the ability to print the assembly instructions 738 for user-defined instructions 736. In order to do this, the debugger 730 must decode machine instructions 740 into assembly instructions 738. This is the same mechanism used by the simulator 712 to decode instructions, and the debugger 730 preferably uses the same DLL used by the simulator 712 to do the decoding. In addition to decoding the instructions, the debugger must convert the decoded instruction into strings. For this purpose, the decode DLL 748 includes a function to map each internal opcode representation to the corresponding mnemonic string. This can be implemented with a simple table.

The user can invoke the prebuilt debugger with a flag or environment variable pointing to the directory containing the user-defined enhancements 750. The prebuilt debugger dynamically opens the appropriate DLL 748.

The debugger 730 also interacts with user-defined state 752. The debugger 730 must be able to read and modify that state 752. In order to do so the debugger 730 communicates with the simulator 712. It asks the simulator 712 how large the state is and what are the names of the state variables. Whenever the debugger 730 is asked to print the value of some user state, it asks the simulator 712 the value in the same way that it asks for predefined state. Similarly, to modify user state, the debugger 730 tells the simulator 712 to set the state to a given value.

Thus, it is seen that implementation of support for user-defined instruction sets and state according to the present invention can be accomplished using modules defining the user functionality which are plugged-in to core software development tools. Thus, a system can be developed in which the plug-in modules for a particular set of user-defined enhancements are maintained as a group within the system for ease of organization and manipulation.

Further, the core software development tools may be specific to particular core instruction sets and processor states, and a single set of plug-in modules for user-defined enhancements may be evaluated in connection with multiple sets of core software development tools resident on the system.

WHAT IS CLAIMED IS:

1. A system for designing a configurable processor, the system comprising:

means for, based on a configuration specification, generating a description of a hardware implementation of the processor; and

means for, based on the configuration specification, generating software development tools specific to the hardware implementation.

2. The system of claim 1, wherein the means for generating software development tools comprises means for generating software development tools capable of generating code to run on the processor.

3. The system of claim 1, wherein the software development tools include a compiler, tailored to the configuration specification, for compiling an application into code executable by the processor.

4. The system of claim 1, wherein the software development tools include an assembler, tailored to the configuration specification, for assembling an application into code executable by the processor.

5. The system of claim 1, wherein the software development tools include a linker, tailored to the configuration specification, for linking code executable by the processor.

6. The system of claim 1, wherein the software development tools include a disassembler, tailored to the configuration specification, for disassembling code executable by the processor.

7. The system of claim 1, wherein the software development tools include a debugger, tailored to the configuration specification, for debugging code executable by the processor.

8. The system of claim 7, wherein the debugger has a common interface and configuration for instruction set simulator and hardware implementations.

9. The system of claim 1, wherein the software development tools include an instruction set simulator, tailored to the configuration specification, for simulating code executable by the processor.

10. The system of claim 9, wherein the instruction set simulator is capable of modeling execution of code being simulated to measure key performance criteria including cycles of execution.

11. The system of claim 10, wherein the performance criteria are based on specific configurable microarchitectural features.



12. The system of claim 10, wherein the instruction set simulator is capable of profiling execution of the program being simulated to record standard profiling statistics, including a number of cycles executed in each simulated function.

13. The system of claim 1, wherein the hardware implementation description includes at least one of a detailed HDL hardware implementation description; synthesis scripts; place and route scripts; programmable logic device scripts; a test bench; diagnostic test for verification; scripts for running diagnostic tests on a simulator; and test tools.

14. The system of claim 1, wherein the means for generating the hardware implementation description comprises:

means for generating a hardware description language description of the hardware implementation description from the configuration specification;

means for synthesizing logic for the hardware implementation based on the hardware description language description; and

means for placing and routing components on a chip based on the synthesized logic to form a circuit.

15. The system of claim 14, the means for generating the hardware implementation description further comprising:

means for verifying timing of the circuit; and

means for determining the area, cycle time and power dissipation of the circuit.

16. The system of claim 1, further comprising means for generating the configuration specification.

17. The system of claim 16, wherein the means for generating the configuration specification is responsive to selection of configuration parameters by a user.

18. The system of claim 16, wherein the means for generating the configuration specification is for generating the specification based on design goals for the processor.

19. The system of claim 1, wherein the configuration specification includes at least one parameter specification of a modifiable characteristic of the processor.

20. The system of claim 19, wherein the at least one parameter specification specifies the inclusion of a functional unit, and at least one processor instruction operating the functional unit.

21. The system of claim 19, wherein the at least one parameter specification specifies one of the inclusion, exclusion and features of a structure affecting processor state.

22. The system of claim 21, wherein the structure is a register file and the parameter specification specifies the number of registers in the register file.

23. The system of claim 21, wherein the structure is an instruction cache.

24. The system of claim 21, wherein the structure is a data cache.

25. The system of claim 21, wherein the structure is a write buffer.

26. The system of claim 21, wherein the structure is one of an on-chip ROM and an on-chip RAM.

27. The system of claim 19, wherein the at least one parameter specification specifies a semantic characteristic controlling the interpretation of at least one of data and instructions in the processor.

28. The system of claim 19, wherein the at least one parameter specification specifies an execution characteristic controlling the execution of instructions in the processor.

29. The system of claim 19, wherein the at least one parameter specification specifies debugging characteristics of the processor.

30. The system of claim 19, wherein the configuration specification includes a parameter specification specifying at least one of a selection of a predetermined feature; a size or number of a processor element; and an assignment of a value.

31. The system of claim 1, further comprising means for evaluating suitability of the configuration specification.

32. The system of claim 31, wherein the means for evaluating includes an interactive estimation tool.



33. The system of claim 31, wherein the means for evaluating is for evaluating hardware characteristics of a processor described by the configuration specification.

34. The system of claim 31, wherein the means for evaluating is for evaluating the suitability of the configuration specification based on estimated performance characteristics of the processor.

35. The system of claim 34, further comprising means for providing information enabling modification of the configuration specification based on the estimated performance characteristics.

36. The system of claim 34, wherein the performance characteristics include at least one of area required to implement the processor on a chip, power consumed by the processor and clock speed of the processor.

37. The system of claim 31, wherein the means for evaluating is for evaluating suitability of the configuration specification based on estimated software characteristics of the processor.

38. The system of claim 37, wherein the means for evaluating is for presenting a suitability evaluation to a user interactively by estimating at least one of code size and cycles required to execute a suite of benchmark programs on a processor described by the configuration specification.

39. The system of claim 31, wherein the means for evaluating is for evaluating hardware characteristics and software characteristics of a processor described by the configuration specification.

40. The system of claim 1 wherein the means for generating is further for providing a characterization of hardware performance and cost and software application performance together to facilitate modification the configuration specification.

41. The system of claim 1 wherein the means for generating is further for providing a characterization of hardware performance and cost and software application performance together to facilitate an extension of the configuration specification.

42. The system of claim 1 wherein the means for generating is further for providing a characterization of hardware performance and cost and software application performance together to facilitate modification of the configuration specification, and for providing a characterization of hardware performance and cost and software application performance together to facilitate the description of an extension of the configuration specification.

43. The system of claim 1, further comprising means for generating a configuration of the processor by extension.

44. The system of claim 1, wherein the configuration specification includes at least one extension specification of an extensible characteristic of the processor.

45. The system of claim 44, wherein the extension specification specifies an additional instruction.

46. The system of claim 44, wherein the extension specification specifies inclusion of a user-defined instruction and an implementation for the instruction.

47. The system of claim 46, wherein the means for generating the software development tools includes means for suggesting to the user potential user-defined instructions particularly suited to at least one application.

48. The system of claim 46, wherein the software development tools include a compiler capable of generating the user-defined instruction.

49. The system of claim 48, wherein the compiler is capable of optimizing code containing user-defined instructions.

50. The system of claim 46, wherein the software development tools include at least one of an assembler capable of generating the user-defined instruction; a simulator capable of simulating execution of user code using the user-defined instruction; and tools capable of verifying the user implementation of the user-defined instruction.

51. The system of claim 45, wherein the compiler is capable of automatically generating additional instructions.

52. The system of claim 44, wherein:

the extension specification specifies a new feature having functionality substantially designed by a user in abstract form; and

the means for generating the hardware implementation description is further for redefining and integrating the new feature into the detailed hardware implementation description.



53. The system of claim 52, wherein the extension specification is a statement in an instruction set architecture language specifying an opcode assignment and an instruction semantic.

54. The system of claim 53, wherein the means for generating the hardware implementation description includes means for generating instruction decode logic from the instruction set architecture language definition.

55. The system of claim 54, wherein the means for generating the hardware implementation description further includes means for generating signals specifying register operand usage for instruction interlock and stall logic based on the instruction set architecture language definition.

56. The system of claim 52, wherein the means for generating software development tools includes means for generating an instruction decode process used in an instruction set simulator tailored to the configuration specification.

57. The system of claim 52, wherein the means for generating software development tools includes means for generating encode tables used in an assembler tailored to the configuration specification.

58. The system of claim 52, wherein the means for generating the hardware implementation description is further for generating a description of datapath hardware for the new feature, the datapath hardware being consistent with a particular pipelined architecture of the processor.

59. The system of claim 44, wherein the additional instruction adds no new state to the processor.

60. The system of claim 44, wherein the additional instruction adds state to the processor.

61. The system of claim 1, wherein the configuration specification includes at least a portion specified by an instruction set architecture description language description.

62. The system of claim 61,wherein the means for generating the hardware implementation description comprises means for generating instruction decode logic automatically from the instruction set architecture language description.

63. The system of claim 61, wherein the means for generating software development tools comprises means for generating an assembler core automatically from the instruction set architecture language description.

64. The system of claim 61, wherein the means for generating software development tools comprises means for generating a compiler automatically from the instruction set architecture language description.

65. The system of claim 61, wherein the means for generating software development tools comprises means for generating a disassembler automatically from the instruction set architecture language description.

66. The system of claim 61, wherein the means for generating software development tools comprises means for generating an instruction set simulator automatically from the instruction set architecture language description.

67. The system of claim 1, wherein the means for generating the hardware implementation description includes means for preprocessing a portion of at least one of the hardware implementation description and the software development tools to modify the hardware implementation description and the software tools, respectively, based on the configuration specification.

68. The system of claim 67, wherein the means for preprocessing is for evaluating an expression in one of the hardware implementation description and the software development tools and replacing the expression with a value based on the configuration specification.

69. The system of claim 68, where the expression includes at least one of an iterative construct, a conditional construct and a database query.

70. The system of claim 1, wherein the configuration specification includes at least one parameter specification specifying a modifiable characteristic of the processor and at least one extension specification specifying an extensible characteristic of the processor.

71. The system of claim 70, wherein the modifiable characteristic is one of a modification to the core specification and an optional feature not specified in the core specification.

72. The system of claim 1, wherein the configuration specification includes at least one parameter specification specifying a binary selectable characteristic of the processor, at least one parametrically specifiable characteristic of the processor, and at least one extension specification specifying an extensible characteristic of the processor.



73. A method of designing a configurable processor, the method comprising:

generating a description of a hardware implementation of the processor based on a configuration specification; and

generating software development tools specific to the hardware implementation based on the configuration specification.

74. A system for designing a configurable processor, the system comprising:

means for generating a configuration specification having a user-definable portion, the user-definable portion of the configuration specification including

a specification of user-defined processor state, and

at least one user-defined instruction and a user-defined function associated therewith, the function including at least one of reading from and writing to the user-defined processor state; and

means for, based on a configuration specification, generating a description of a hardware implementation of the processor.

75. The system of claim 74, wherein the description of the hardware implementation of the processor includes a description of control logic necessary for execution of the at least one user-defined instruction and for implementation of the user-defined processor state.

76. The system of claim 75, wherein:

the hardware implementation of the processor describes an instruction execution pipeline; and

the control logic includes portions associated with each stage of the instruction execution pipeline.

77. The system of claim 76, wherein:

the hardware implementation description includes a description of circuitry for aborting instruction execution; and

the control logic includes circuitry for preventing modification of the user-defined state by aborted instructions.

78. The system of claim 77, wherein the control logic includes circuitry for performing at least one of an instruction issue, an operand bypass and an operand write enable for the at least one user-defined instruction.

79. The system of claim 76, wherein the hardware implementation description includes registers for implementing the user-defined state in a plurality of stages of the instruction execution pipeline.

80. The system of claim 76, wherein:

the hardware implementation description includes state registers written in a different pipeline stage than one in which output operands are produced; and

the hardware implementation description specifies that such writes are bypassed into subsequent instructions that reference the user-defined processor state before writes to the state are committed.

81. The system of claim 74, wherein;

the configuration specification includes a predetermined portion in addition to the user-defined portion; and

the predetermined portion of the specification includes an instruction for facilitating saving the user-defined state to memory and an instruction for facilitating restoring the user-defined state from memory.

82. The system of claim 81, further comprising means for generating software to context switch the user-defined state using the instruction.

83. The system of claim 74, further comprising means for producing at least one of an assembler for assembling the user-defined processor state and the at least one user-defined instruction; a compiler for compiling the user-defined processor state and the at least one user-defined instruction; a simulator for simulating the user-defined processor state and the at least one user-defined instruction; and a debugger for debugging the user-defined processor state and the at least one user-defined instruction.

84. The system of claim 74, further comprising means for producing an assembler for assembling the user-defined processor state and the at least one user-defined instruction, a compiler for compiling the user-defined processor state and the at least one user-defined instruction, a simulator for simulating the user-defined processor state and the at least one user-defined instruction and a debugger for debugging the user-defined processor state and the at least one user-defined instruction.

85. The system of claim 74, wherein the user-defined portion of the specification includes at least one statement specifying a size and indexing of the user-defined state.

86. The system of claim 85, wherein the user-defined portion of the specification includes at least one attribute associated with the user-defined state and specifying packing of the user-defined state in a processor register.

87. The system of claim 74, wherein the user-defined portion of the specification includes at least one statement specifying a mapping of the user-defined state to processor registers.



88. The system of claim 74, wherein the means for generating the hardware implementation description includes means for automatically mapping the user-defined state to processor registers.

89. The system of claim 74, wherein the user-defined portion of the specification includes at least one statement specifying a class of user-defined instructions and its effect on the user-defined state.

90. The system of claim 74, wherein the user-defined portion of the specification includes at least one assignment statement assigning a value to the user-defined state.

91. A system for designing a configurable processor, the system comprising:

core software tools for, based on an instruction set architecture specification, generating software development tools specific to the specification; and

a user-defined instruction module for, based on a user-defined instruction specification, generating at least one module for use by the core software tools in implementing the user-defined instructions.

92. The system of claim 91, wherein the core software tools comprise software tools capable of generating code to run on the processor.

93. The system of claim 91, wherein the at least one module is implemented as a dynamically linked library.

94. The system of claim 91, wherein the at least one module is implemented as a table.

95. The system of claim 91, wherein the core software tools include a compiler for, using the user-defined instruction module, compiling an application into code using the user-defined instructions and executable by the processor.

96. The system of claim 95, wherein the at least one module includes a module for use by the compiler in compiling the user-defined instructions.

97. The system of claim 91, wherein the core software tools include an assembler for using the user-defined module to assemble an application into code using the user-defined instructions and executable by the processor.

98. The system of claim 97, wherein the at least one module includes a module for use by the assembler in mapping assembly language instructions to the user-defined instructions.

99. The system of claim 98, wherein:

the system further includes a core instruction set specification specifying non-user defined instructions; and

the core instruction set specification is used by the assembler to assemble the application into code executable by the processor.

100. The system of claim 91, wherein the core software tools include an instruction set simulator for simulating code executable by the processor.

101. The system of claim 100, wherein the at least one module includes a simulator module for use by the simulator in simulating execution of the user-defined instructions.

102. The system of claim 101, wherein the module for use by the simulator includes data for decoding the user-defined instructions.

103. The system of claim 102, wherein the simulator uses a module to decode instructions using the simulator module when they cannot be decoded as predefined instructions.

104. The system of claim 91, wherein the core software tools include a debugger for using the user-defined module to debug code using the user-defined instructions and executable by the processor.

105. The system of claim 104, wherein the at least one module includes a module usable by the debugger to decode machine instructions into assembly instructions.

106. The system of claim 104, wherein the at least one module includes a module usable by the debugger to convert assembly instructions into strings.

107. The system of claim 104, wherein:

the core software tools include an instruction set simulator for simulating code executable by the processor; and

the debugger is for communicating with the simulator to obtain information on the user-defined state for debugging.

108. The system of claim 91, wherein a single user-defined instruction can be used unmodified by multiple core software tools based on different core instruction set specifications.



109. A system for designing a configurable processor, the system comprising:

core software tools for, based on an instruction set architecture specification, generating software development tools specific to the specification;

a user-defined instruction module for, based on a user-defined instruction specification, generating a group of at least one module for use by the core software tools in implementing the user-defined instructions; and

storage means for concurrently storing groups generated by the user-defined instruction module, each of the groups corresponding to a different set of user-defined instructions.

110. The system of claim 109, wherein the at least one module is implemented as a dynamically linked library.

111. The system of claim 109, wherein the at least one module is implemented as a table.

112. The system of claim 109, wherein the core software tools include a compiler for, using the user-defined instruction module, compiling an application into code using the user-defined instructions and executable by the processor.

113. The system of claim 112, wherein the at least one module includes a module for use by the compiler in compiling the user-defined instructions.

114. The system of claim 109, wherein the core software tools include an assembler for using the user-defined module to assemble an application into code using the user-defined instructions and executable by the processor.

115. The system of claim 114, wherein the at least one module includes a module for use by the assembler in mapping assembly language instructions to the user-defined instructions.

116. The system of claim 109, wherein the core software tools include an instruction set simulator for simulating code executable by the processor.

117. The system of claim 116, wherein the at least one module includes a module for use by the simulator in simulating execution of the user-defined instructions.

118. The system of claim 117, wherein the module for use by the simulator includes data for decoding the user-defined instructions.

119. The system of claim 118, wherein the simulator uses a module to decode instructions using the simulator module when they cannot be decoded as predefined instructions.

120. The system of claim 109, wherein the core software tools include a debugger for using the user-defined module to debug code using the user-defined instructions and executable by the processor.

121. The system of claim 120, wherein the at least one module includes a module usable by the debugger to decode machine instructions into assembly instructions.

122. The system of claim 120, wherein the at least one module includes a module usable by the debugger to convert assembly instructions into strings.

123. A system for designing a configurable processor, the system comprising:

a plurality of groups of core software tools, each group for, based on an instruction set architecture specification, generating software development tools specific to the specification; and

a user-defined instruction module for, based on a user-defined instruction specification, generating at least one module for use by a group of core software tools in implementing the user-defined instructions.

124. The system of claim 123, wherein the at least one module is implemented as a dynamically linked library.

125. The system of claim 123, wherein the at least one module is implemented as a table.

126. The system of claim 123, wherein at least one group of core software tools include a compiler for, using the user-defined instruction module, compiling an application into code using the user-defined instructions and executable by the processor.

127. The system of claim 126, wherein the at least one module includes a module for use by the compiler in compiling the user-defined instructions.

128. The system of claim 123, wherein at least one group of core software tools include an assembler for using the user-defined module to assemble an application into code using the user-defined instructions and executable by the processor.

129. The system of claim 128, wherein the at least one module includes a module for use by the assembler in mapping assembly language instructions to the user-defined instructions.



130. The system of claim 123, wherein at least one group of core software tools includes an instruction set simulator for simulating code executable by the processor.

131. The system of claim 130, wherein the at least one module includes a module for use by the simulator in simulating execution of the user-defined instructions.

132. The system of claim 131, wherein the module for use by the simulator includes data for decoding the user-defined instructions.

133. The system of claim 132, wherein the simulator uses a module to decode instructions using the simulator module when they cannot be decoded as predefined instructions.

134. The system of claim 123, wherein at least one group of core software tools include a debugger for using the user-defined module to debug code using the user-defined instructions and executable by the processor.

135. The system of claim 134, wherein the at least one module includes a module usable by the debugger uses to decode machine instructions into assembly instructions.

136. The system of claim 134, wherein the at least one module includes a module usable by the debugger to convert assembly instructions into strings.

## ABSTRACT OF THE DISCLOSURE

In a first aspect of the invention, a configurable RISC processor implements an instruction set which provides good code density in a fixed-length high-performance encoding based on RISC principles, including a general register with load/store architecture. Further, the processor implements a simple variable-length encoding that maintains high performance. In a second aspect of the invention, when selecting and building a processor configuration, a user creates a new set of user-defined instructions, places them in a file directory, and invokes a tool that processes the user instructions and transforms them into a form usable by the software development tools. In this way, the user may customize a processor configuration by adding new instructions and within minutes, be able to evaluate that feature. The user is able to keep multiple sets of potential instructions and easily switch between them when evaluating their application. In a third aspect of the invention, an automated processor design tool uses a description of customized processor instruction set extensions in a standardized language to develop a configurable definition of a target instruction set, a Hardware Description Language description of circuitry necessary to implement the instruction set, and development tools which can be used to develop applications for the processor and to verify it. The standardized language is capable of handling instruction set extensions which modify processor state or use configurable processors. By providing a constrained domain of extensions and optimizations, the process can be automated to a high degree, thereby facilitating fast and reliable development.







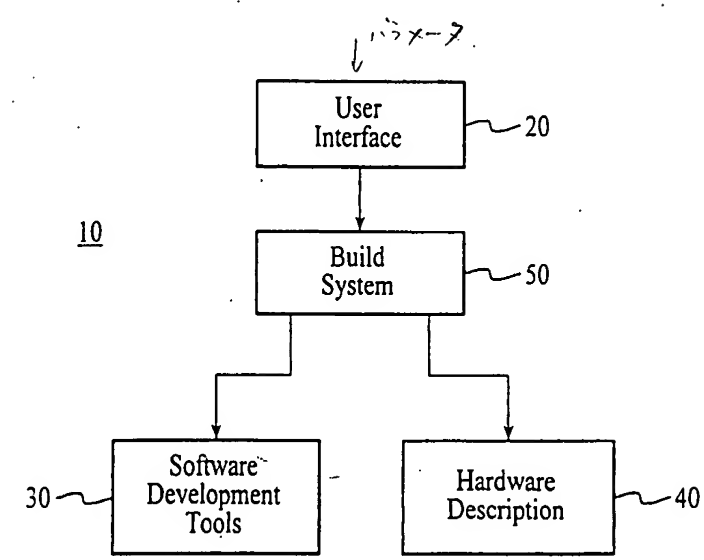



FIG. 1

2/21

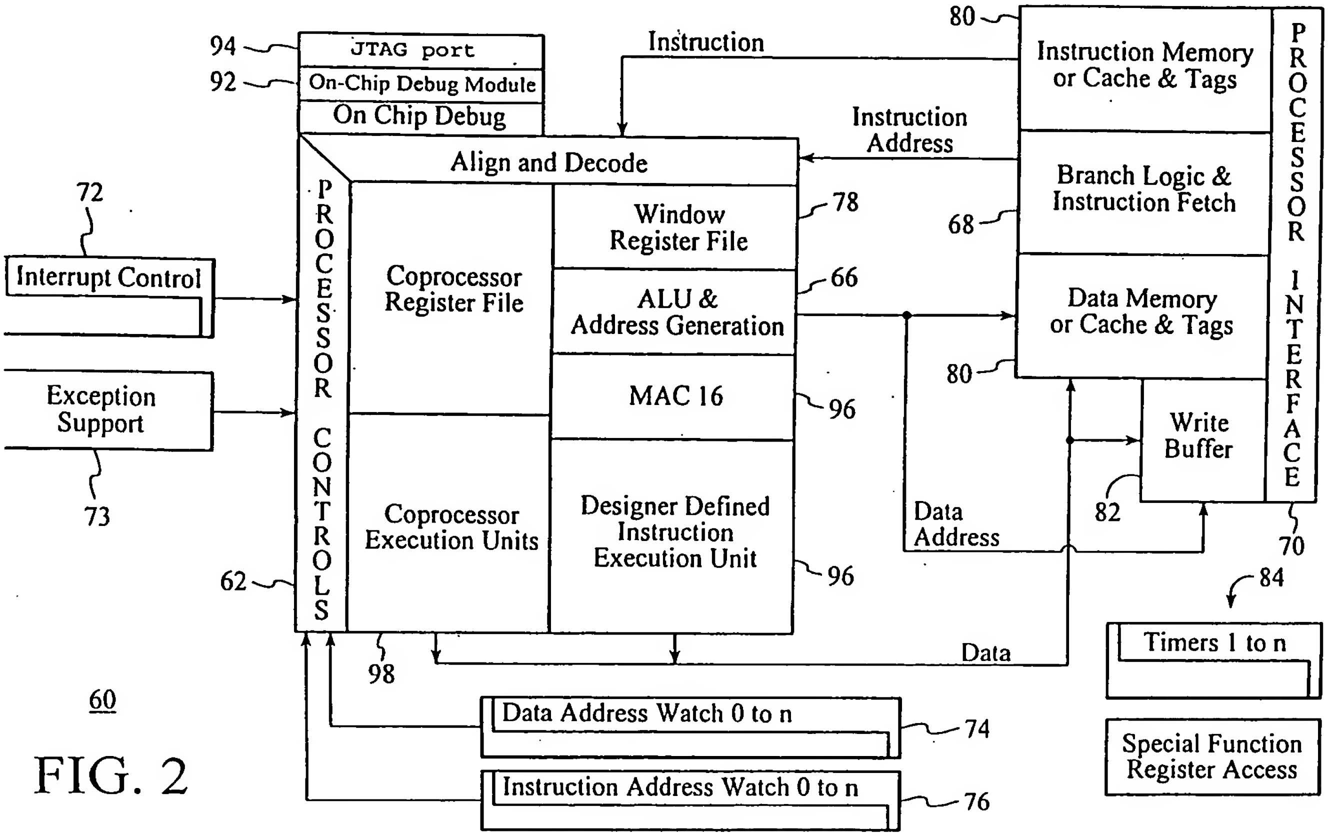



FIG. 2

プロセッサ構成

3/21

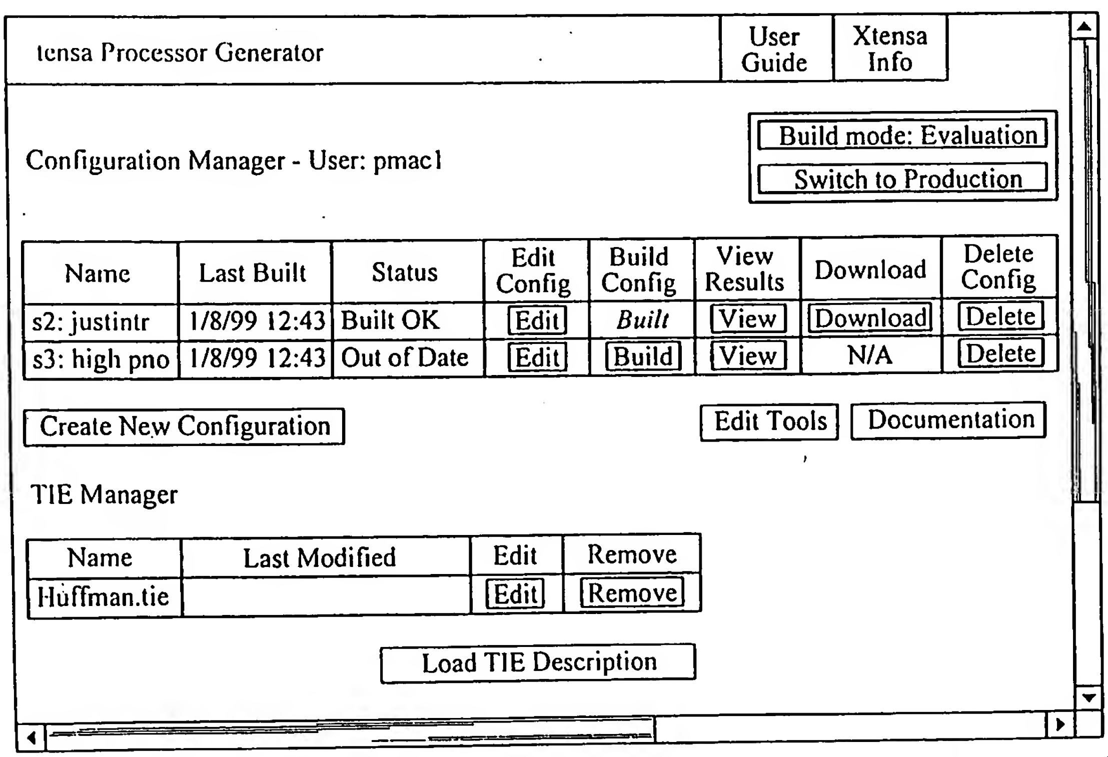
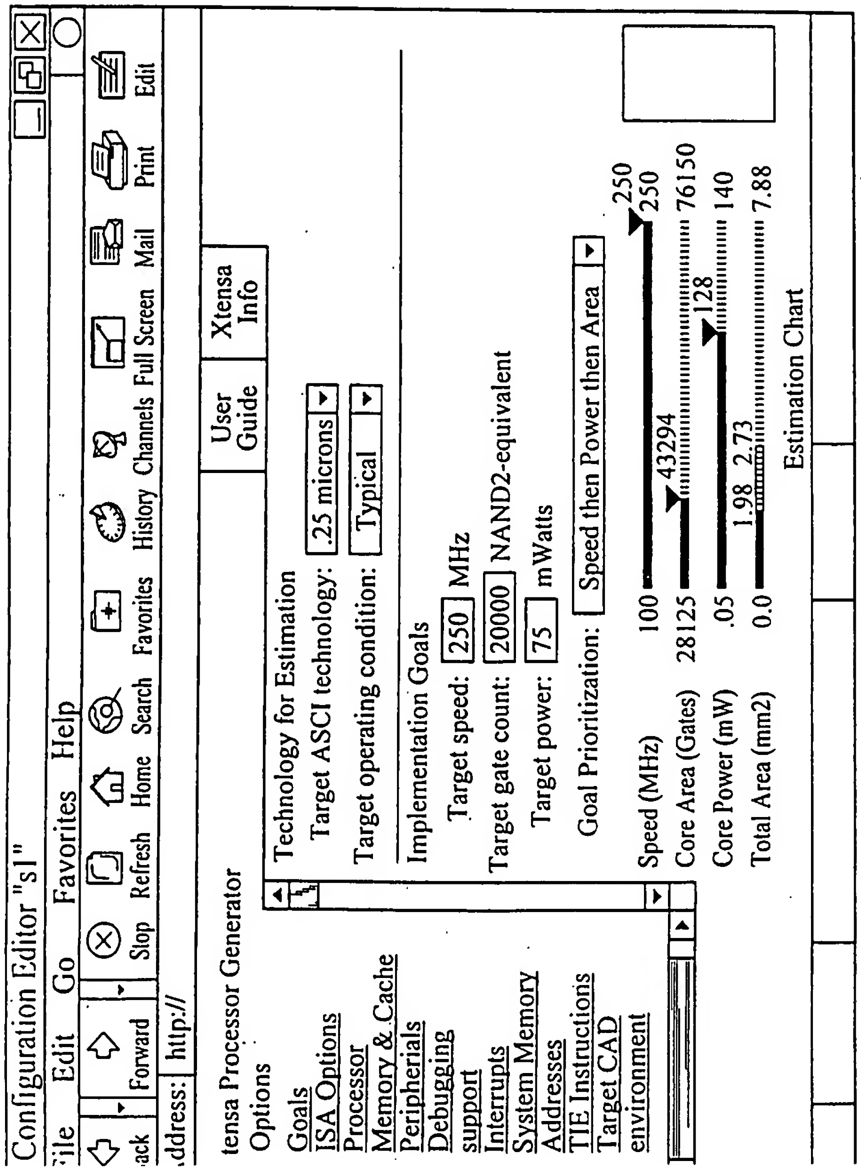



FIG. 3

86 構成管理画面


Configuration Editor "s1"
File Edit Go Favorites Help
Back Forward Stop Refresh Home Search Favorites History Channels Full Screen Mail Print Edit
Address: http://
Xtensa Processor Generator
User Guide
Xtensa Info
Options
Goals
ISA Options
Processor
Memory & Cache
Peripherials
Debugging support
Interrupts
System Memory Addresses
TIE Instructions
Target CAD environment
Technology for Estimation
Target ASCI technology: .25 microns
Target operating condition: Typical
Implementation Goals
Target speed: 250 MHz
Target gate count: 20000 NAND2-equivalent
Target power: 75 mWatts
Goal Prioritization: Speed then Power then Area
Speed (MHz) 100 250 250
Core Area (Gates) 28125 43294 76150
Core Power (mW) .05 128 140
Total Area (mm2) 0.0 1.98 2.73 7.88
Estimation Chart

88

FIG. 4



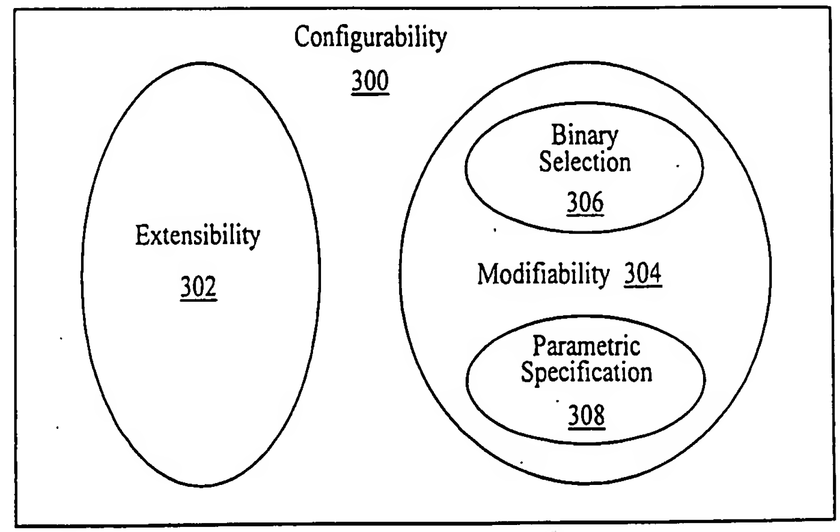
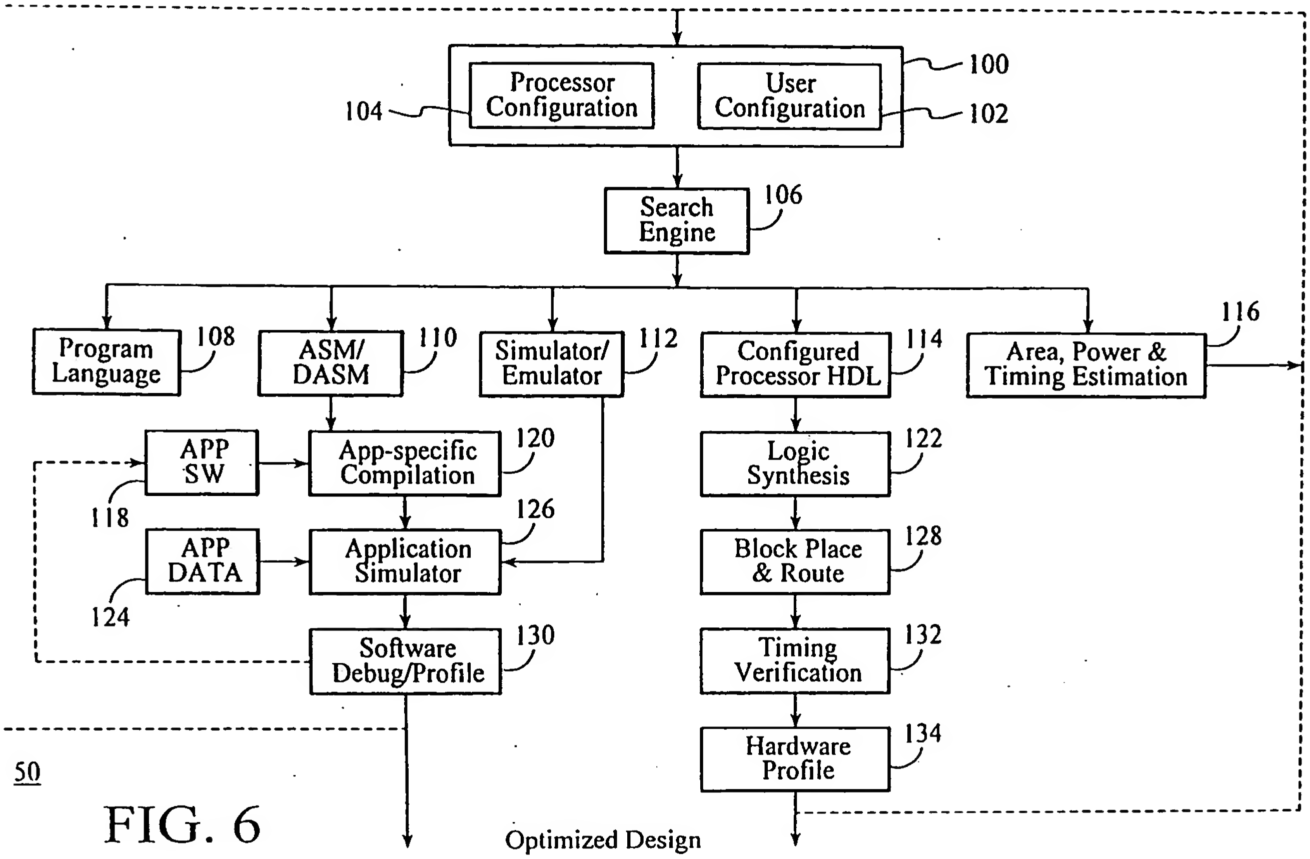

Configurability
300
Extensibility
302
Modifiability 304
Binary Selection
306
Parametric Specification
308

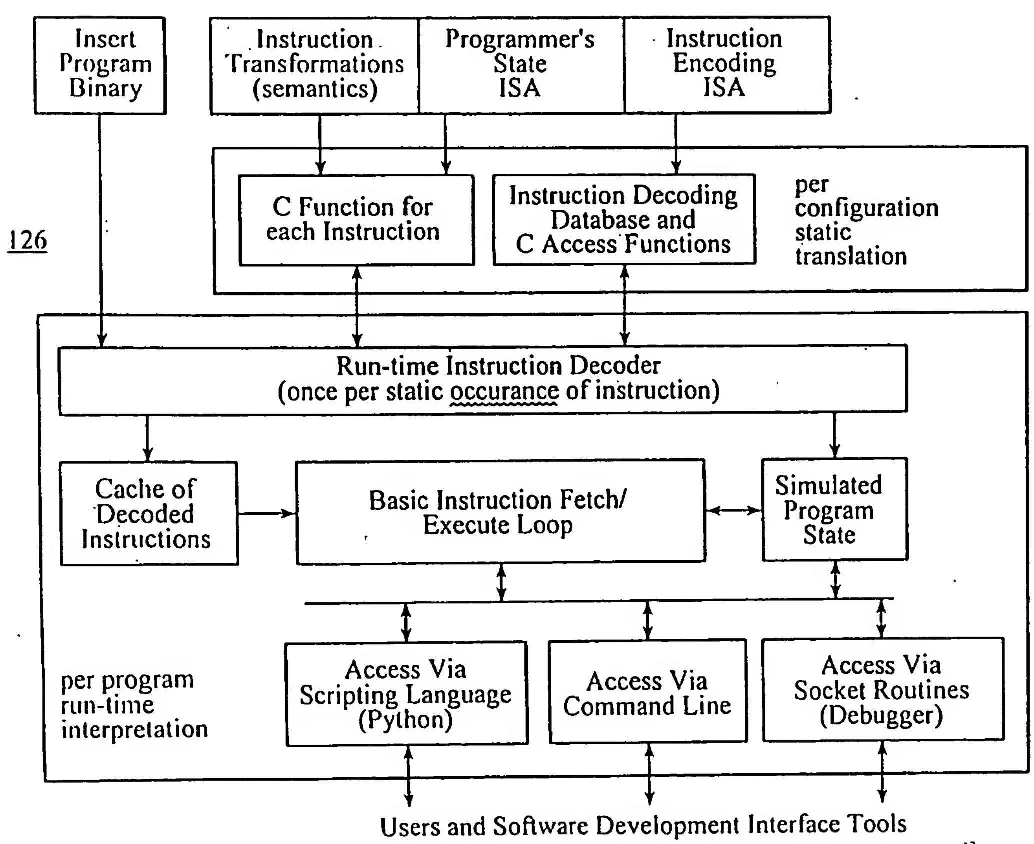
FIG. 5



Processor Configuration
104
User Configuration
100
102
Search Engine
106
Program Language
108
ASM/ DASM
110
Simulator/ Emulator
112
Configured Processor HDL
114
Area, Power & Timing Estimation
116
APP SW
118
App-specific Compilation
120
APP DATA
124
Application Simulator
126
Software Debug/Profile
130
Logic Synthesis
122
Block Place & Route
128
Timing Verification
132
Hardware Profile
134
50
Optimized Design

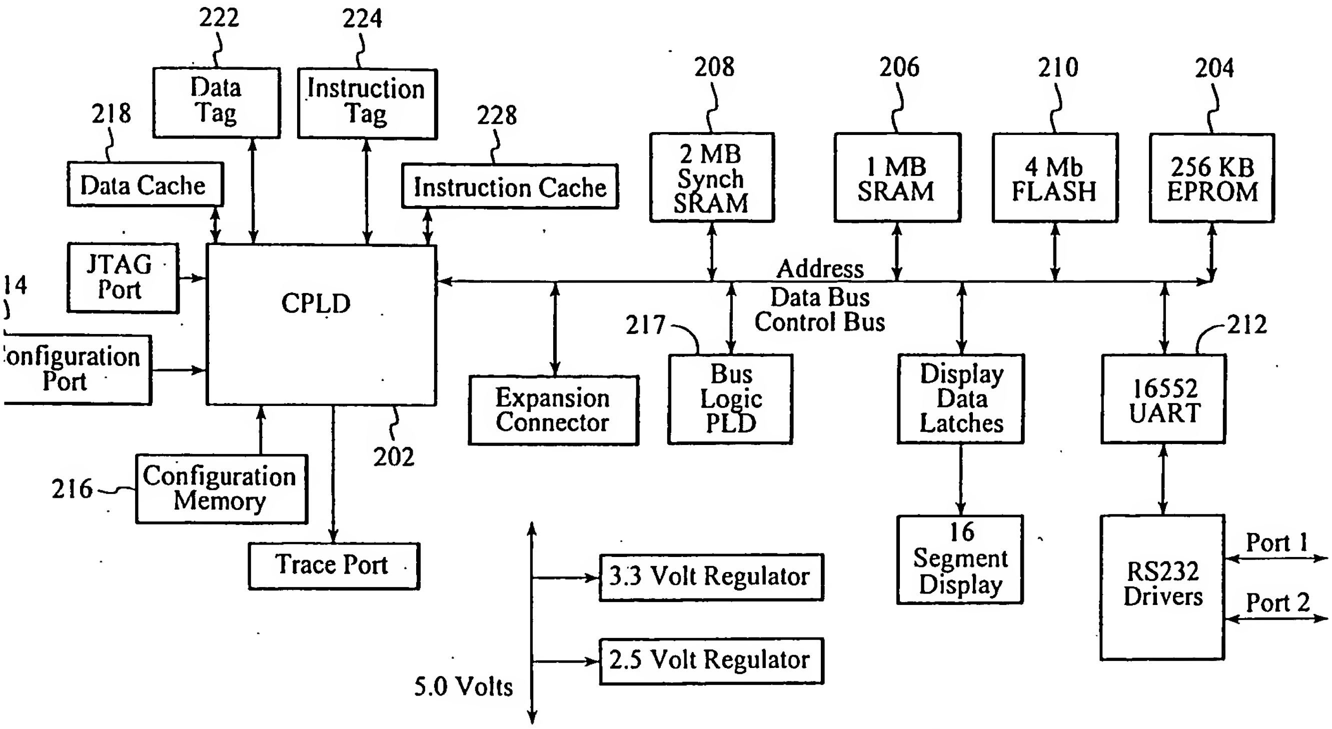

FIG. 6





Insert Program Binary
Instruction Transformations (semantics)
Programmer's State ISA
Instruction Encoding ISA
126
C Function for each Instruction
Instruction Decoding Database and C Access Functions
per configuration static translation
Run-time Instruction Decoder
(once per static occurance of instruction)
Cache of Decoded Instructions
Basic Instruction Fetch/ Execute Loop
Simulated Program State
per program run-time interpretation
Access Via Scripting Language (Python)
Access Via Command Line
Access Via Socket Routines (Debugger)
Users and Software Development Interface Tools


FIG. 7




200

FIG. 8



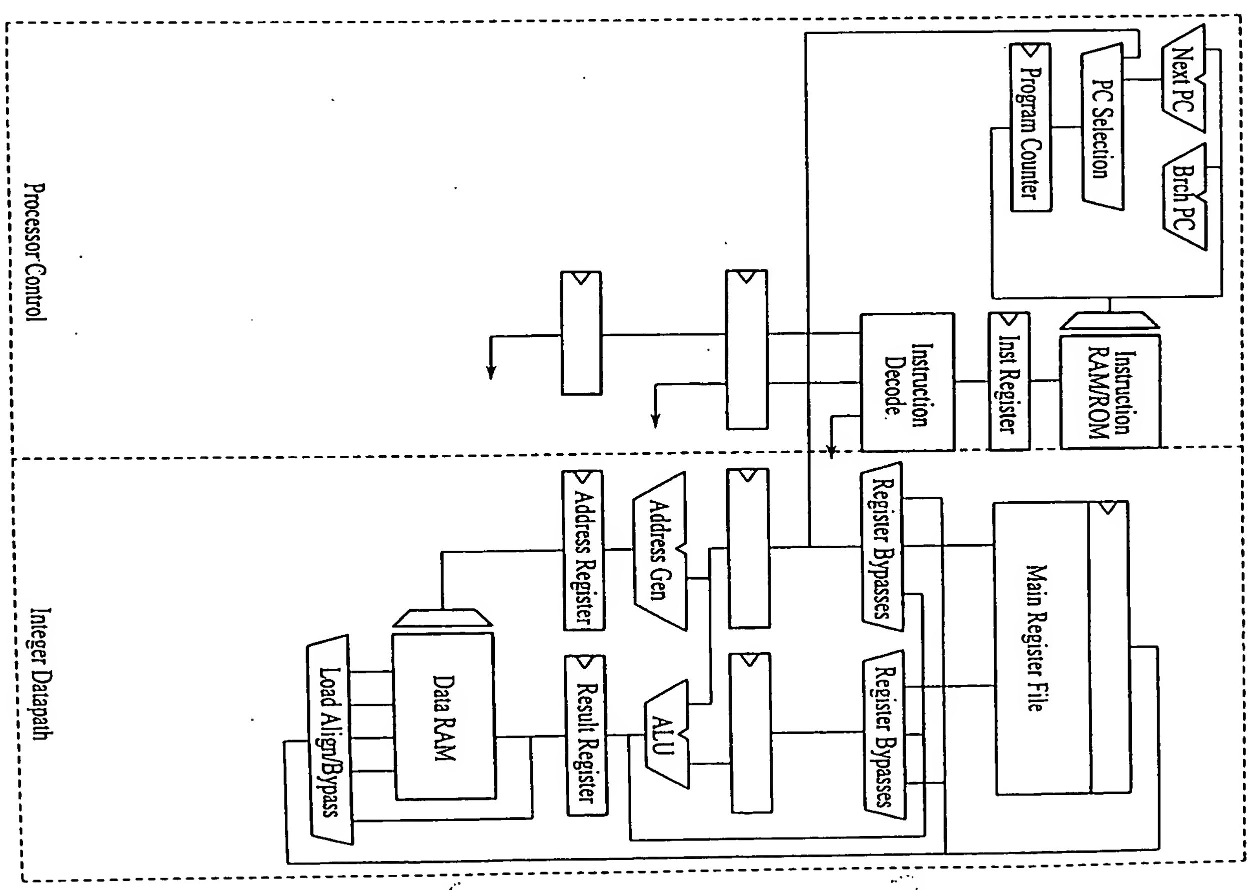



FIG. 9

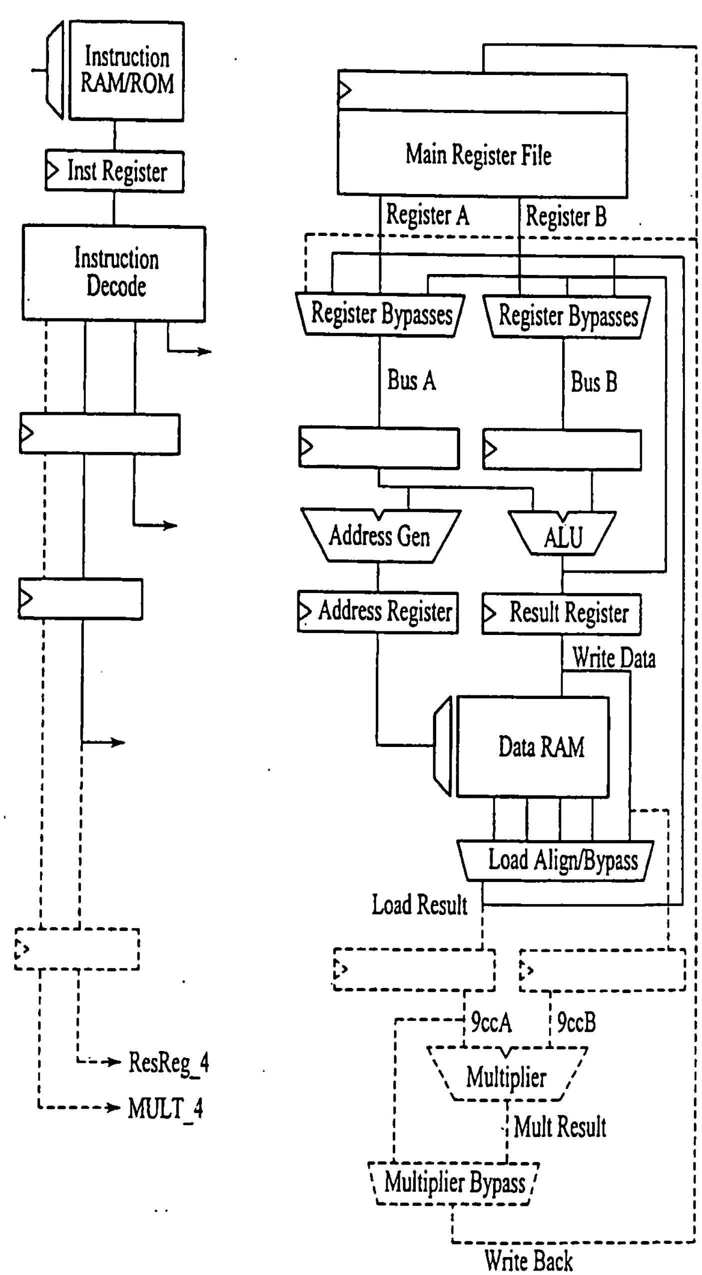

Instruction RAM/ROM
Inst Register
Instruction Decode
ResReg_4
MULT_4
Main Register File
Register A
Register B
Register Bypasses
Register Bypasses
Bus A
Bus B
Address Gen
ALU
Address Register
Result Register
Write Data
Data RAM
Load Align/Bypass
Load Result
9ccA
9ccB
Multiplier
Mult Result
Multiplier Bypass
Write Back

FIG. 10



FIG. 11

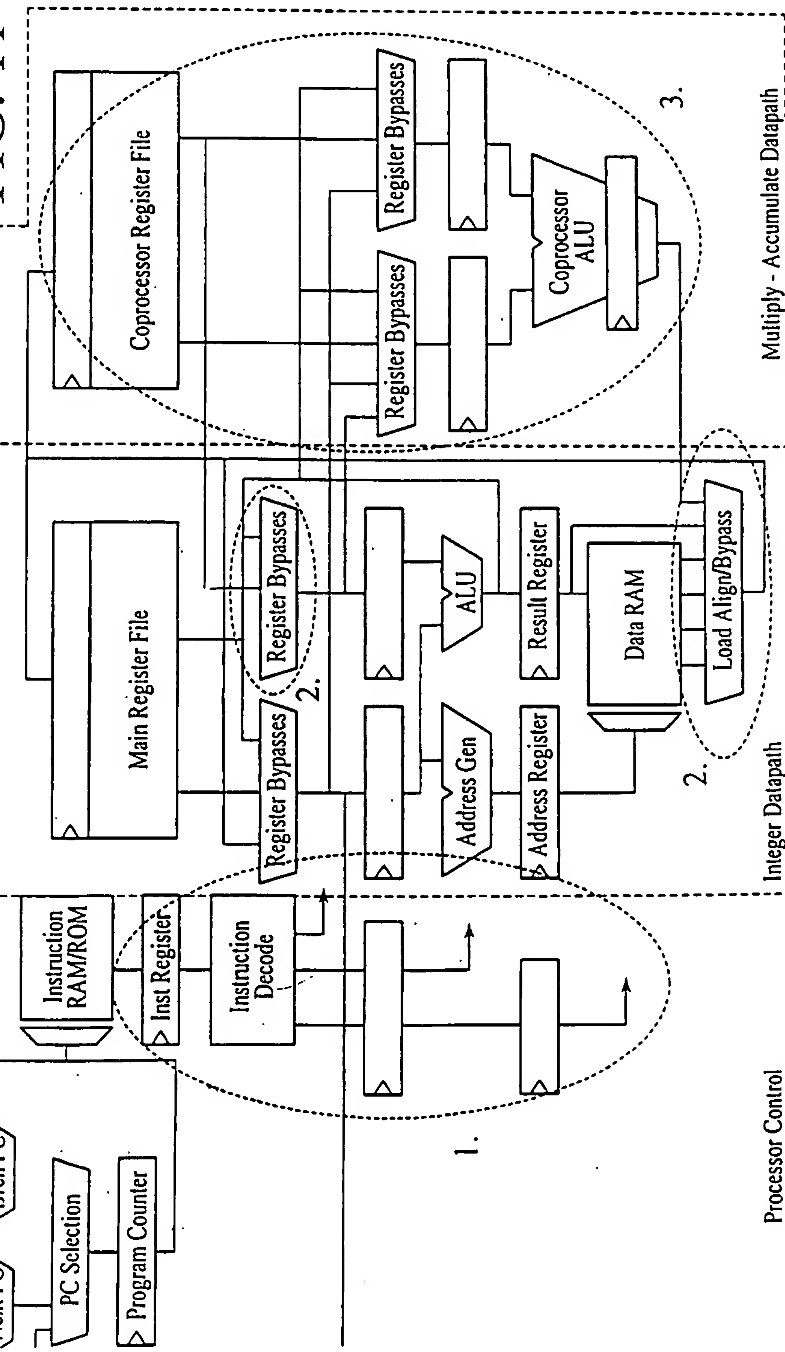

Coprocessor Register File
Register Bypasses
Register Bypasses
Coprocessor ALU
3.
Multiply - Accumulate Datapath
Main Register File
Register Bypasses
Register Bypasses
2.
ALU
Address Gen
Result Register
Address Register
Data RAM
Load Align/Bypass
2.
Integer Datapath
Instruction RAM/ROM
Inst Register
Instruction Decode
1.
Brch PC
Next PC
PC Selection
Program Counter
Processor Control

FIG. 12

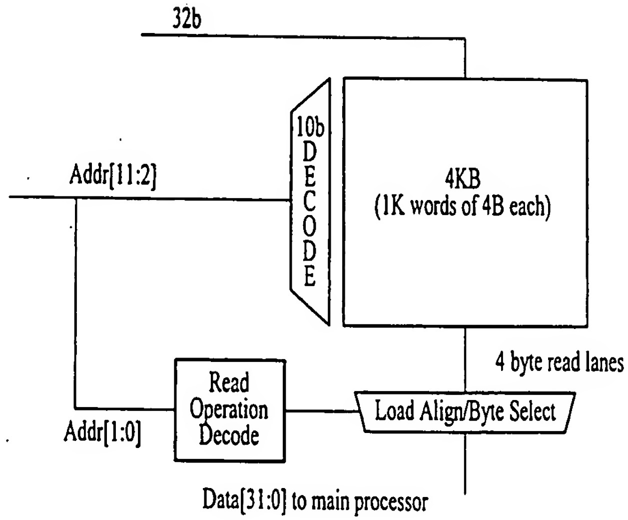
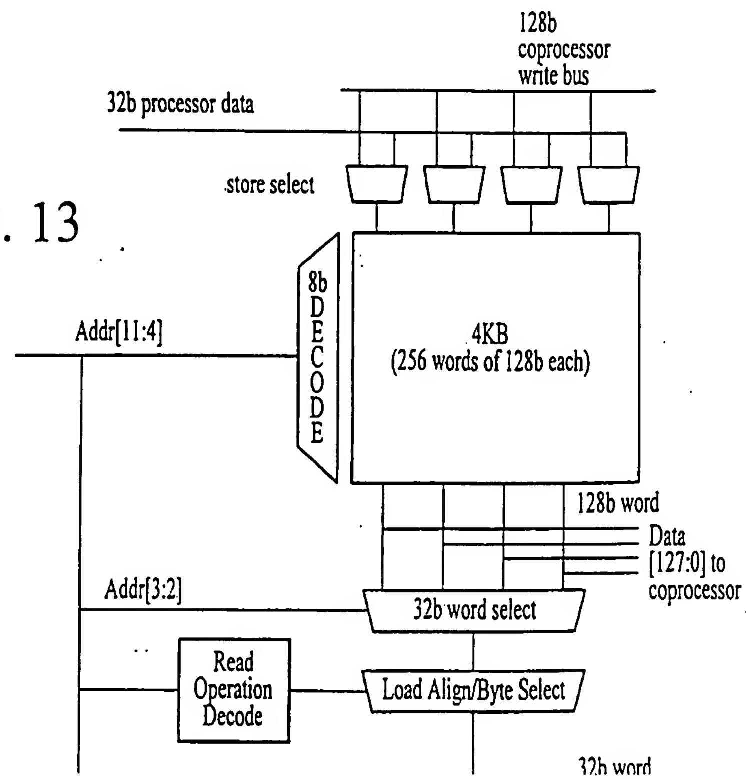



FIG. 13





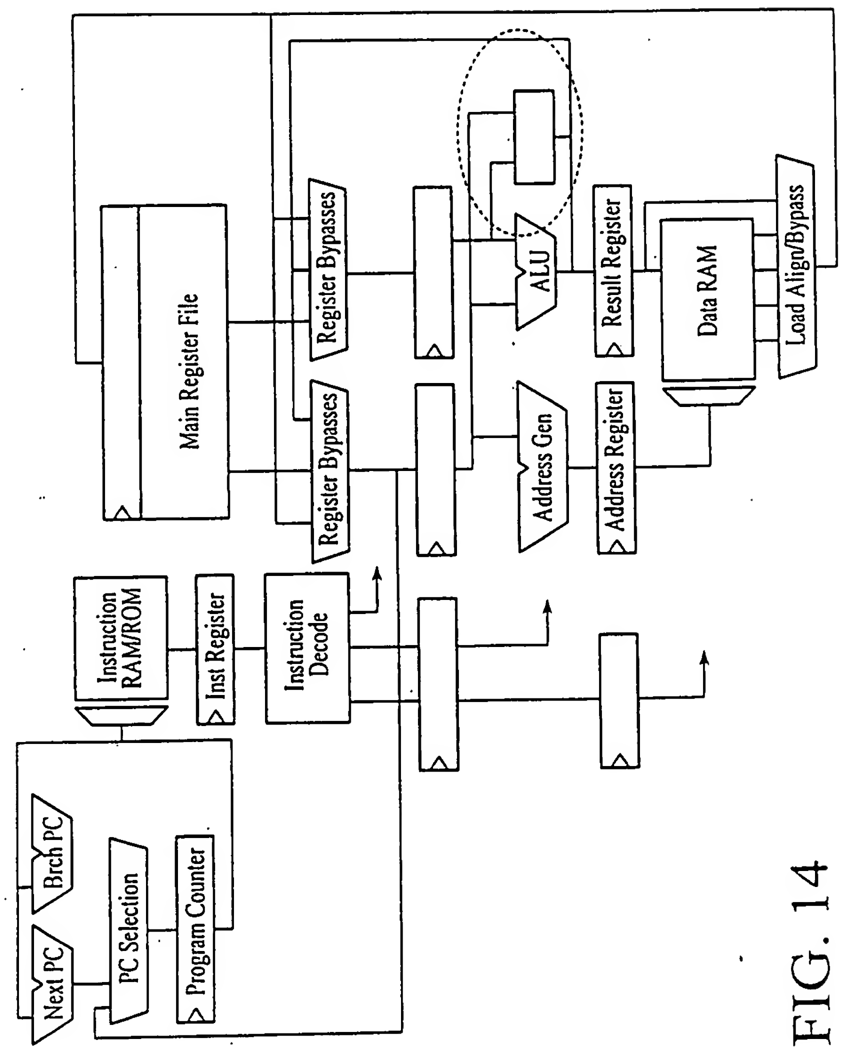




FIG. 14

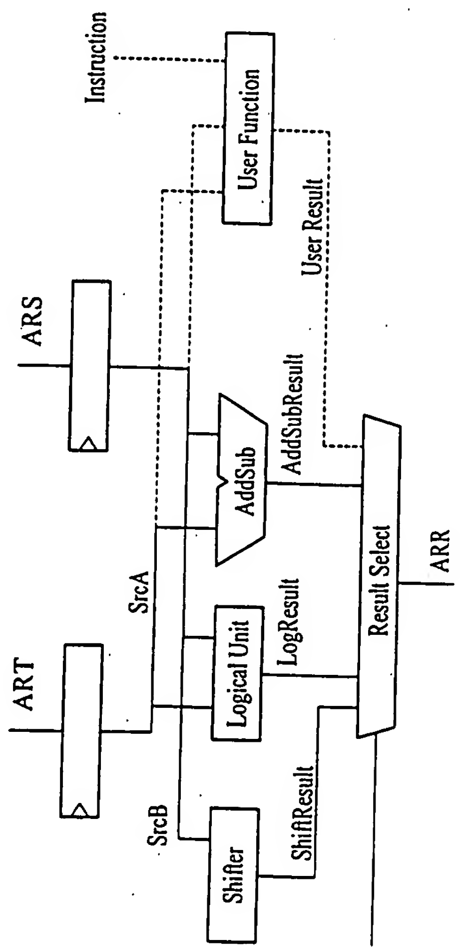

Instruction
User Function
User Result
ARS
AddSub
AddSubResult
SrcA
ART
Logical Unit
LogResult
Result Select
ARR
SrcB
Shifter
ShiftResult

FIG. 15



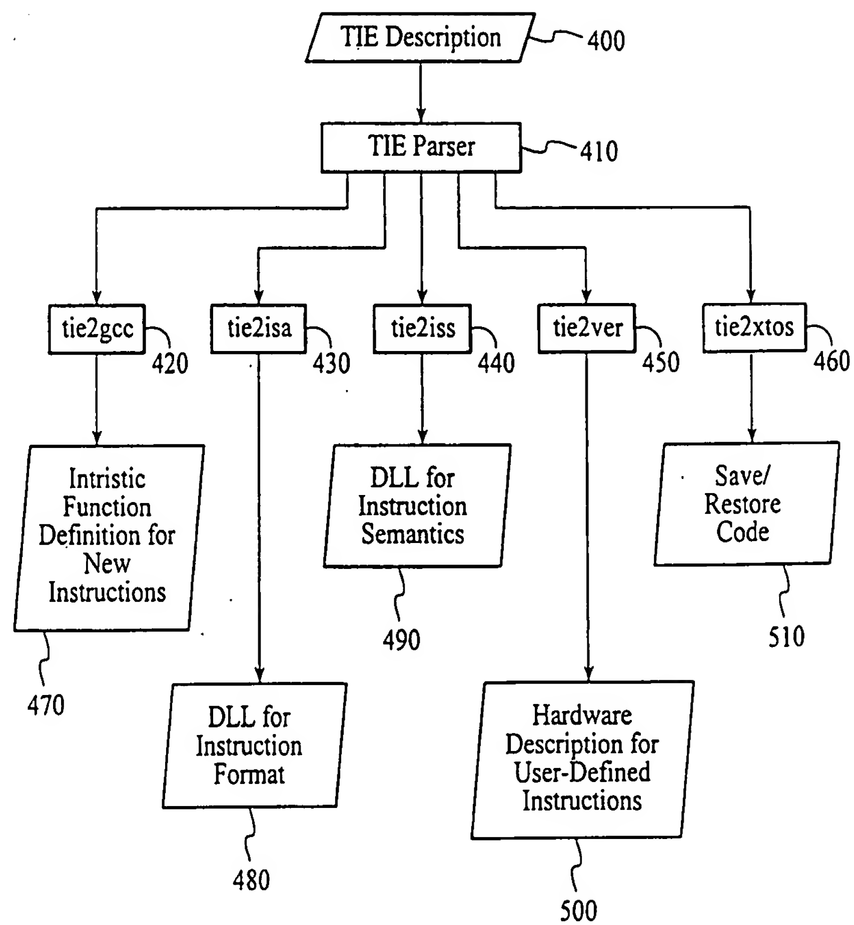

TIE Description
400
TIE Parser
410
tie2gcc
420
tie2isa
430
tie2iss
440
tie2ver
450
tie2xtos
460
Intristic Function Definition for New Instructions
470
DLL for Instruction Format
480
DLL for Instruction Semantics
490
Hardware Description for User-Defined Instructions
500
Save/ Restore Code
510

FIG. 16

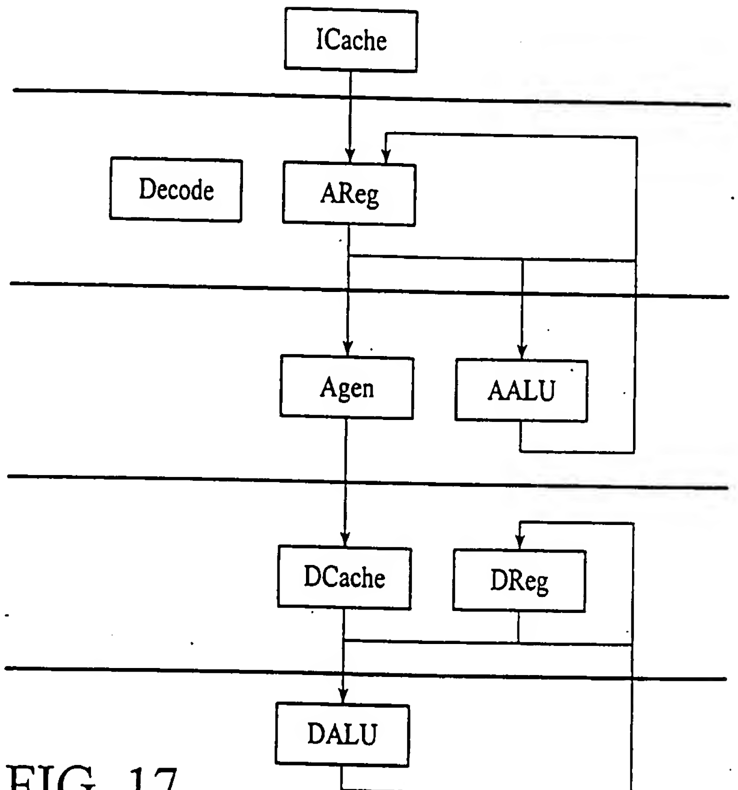



FIG. 17

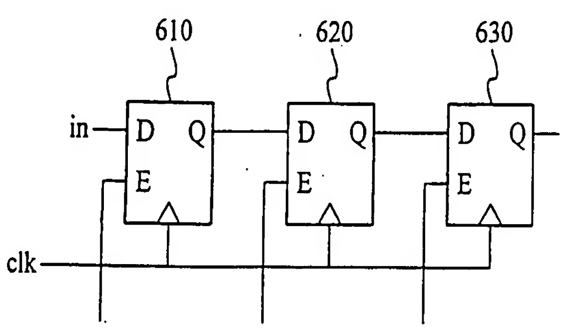



FIG. 18



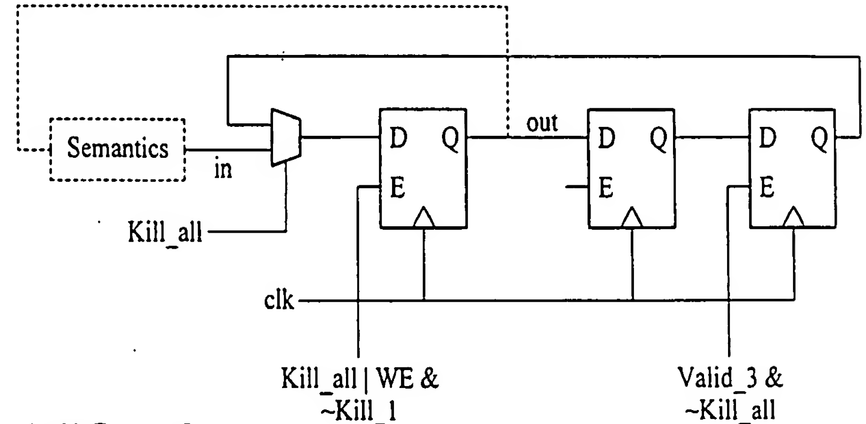



FIG. 19

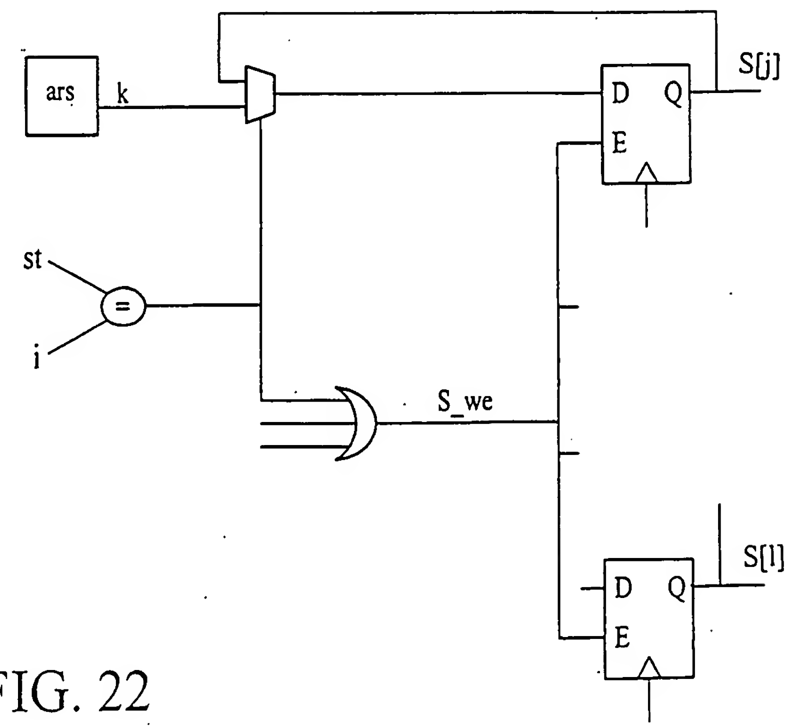



FIG. 22

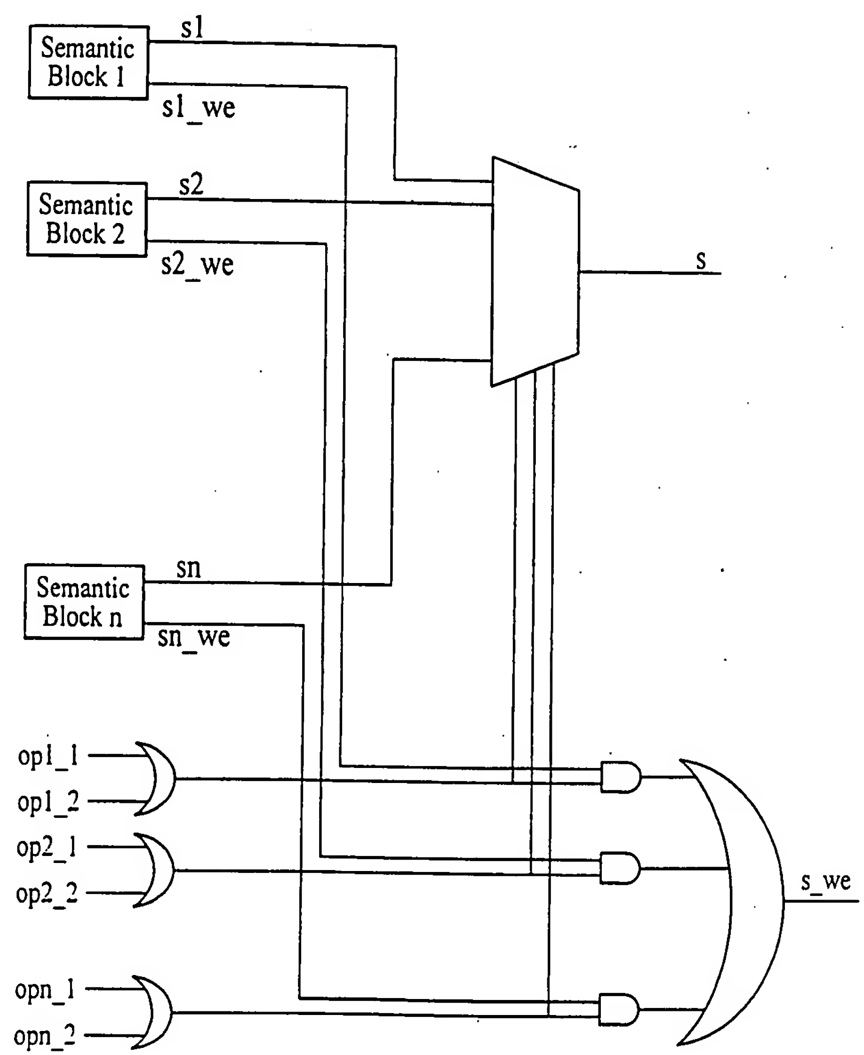

Semantic Block 1
s1
s1_we
Semantic Block 2
s2
s2_we
Semantic Block n
sn
sn_we
s
op1_1
op1_2
op2_1
op2_2
opn_1
opn_2
s_we

FIG. 20



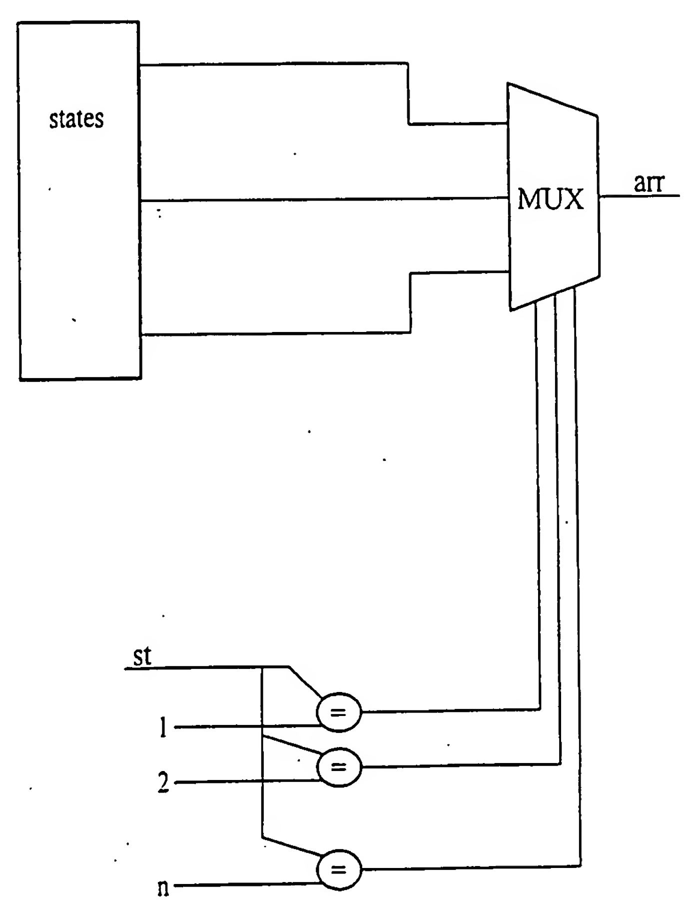

states
MUX
arr
st
1
2
n
=
=
=

FIG. 21

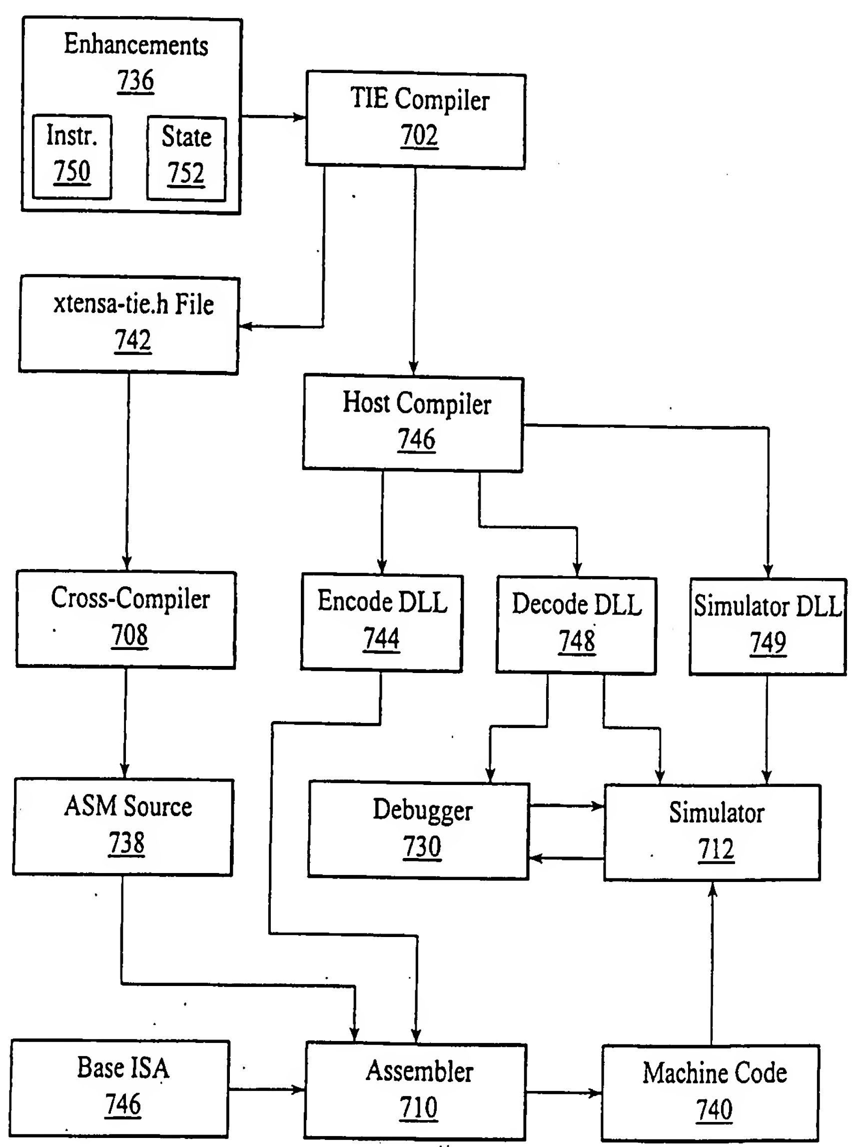

Enhancements
736
Instr.
750
State
752
TIE Compiler
702
xtensa-tie.h File
742
Host Compiler
746
Cross-Compiler
708
Encode DLL
744
Decode DLL
748
Simulator DLL
749
ASM Source
738
Debugger
730
Simulator
712
Base ISA
746
Assembler
710
Machine Code
740

FIG. 23



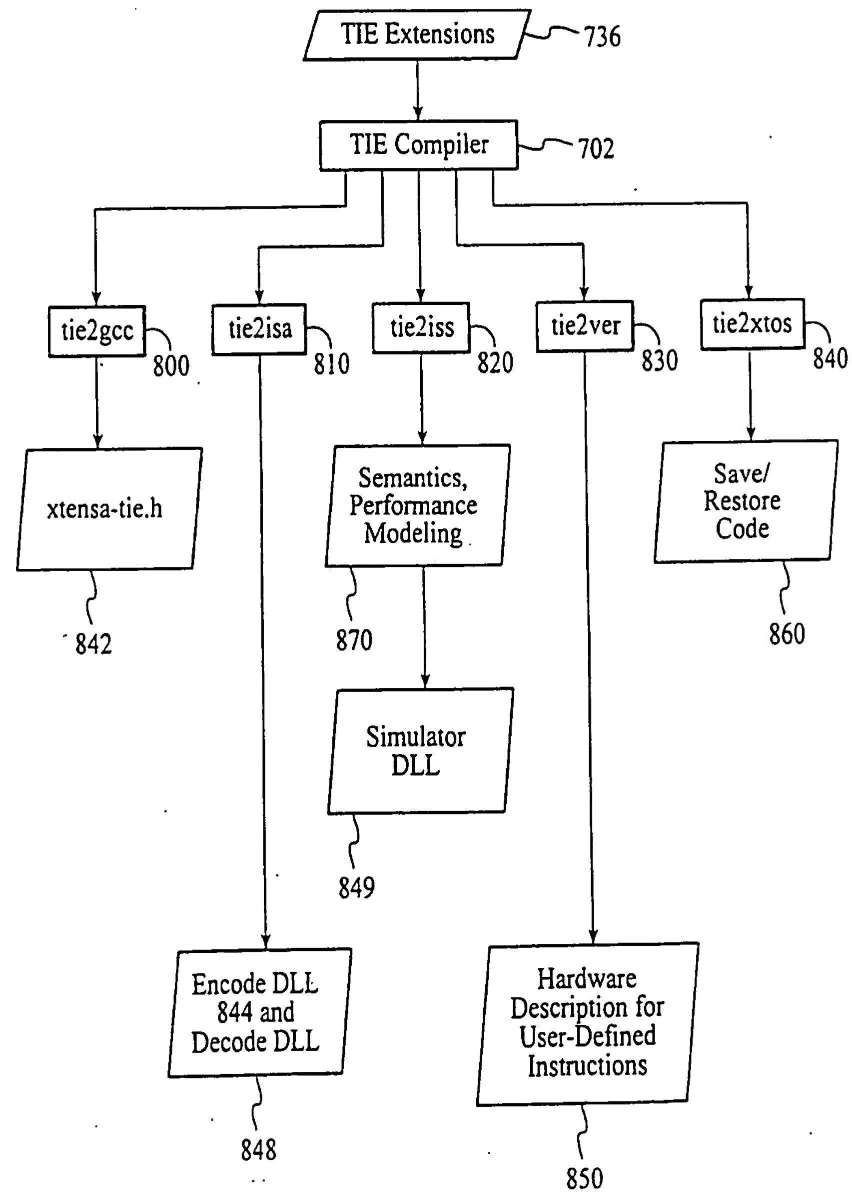

TIE Extensions
736
TIE Compiler
702
tie2gcc
800
tie2isa
810
tie2iss
820
tie2ver
830
tie2xtos
840
xtensa-tie.h
842
Semantics, Performance Modeling
870
Save/ Restore Code
860
Simulator DLL
849
Encode DLL 844 and Decode DLL
848
Hardware Description for User-Defined Instructions
850

FIG. 24